User's Guide

DSP Master BSP

for the UC1394a-1 MCM

Contents

1 PREFACE 9

1.1 Document Organization 9

1.2 Documentation Overview 9

1.3 Notational Conventions 9

1.4 Trademarks 11

1.5 Revision History 11

2 SYSTEM OVERVIEW 12

2.1 Applications 12

2.2 Block Diagram 13

2.3 DSP 14

2.4 FPGA 14

2.5 Peripheral Interface 14

2.6 IEEE1394 Interface 14

2.7 UART interface 14

2.8 I/O Pins 15

2.9 McBSP Interfaces 15

2.10 USB interface 15

2.11 I^2C interface 15

2.12 ADC 15

2.13 RTC 15

2.14 LED 15

2.15 System Reset 16

2.16 DSP JTAG Interface 16

2.17 FPGA JTAG Interface 16

2.18 Power Supply 16

3 QUICK START 17

4 PROGRAMMING THE UC1394A-1 18

4.1 Required Tools 18

4.2 Software Development Flow 18
4.2.1 Module Support Library 19

4.3 Startup Procedure 20

4.4 System Initialization 21

4.5 Main Loop 23

4.6 How to Store an Application in Flash Memory 24

4.7 Hints for Programming the TMS320VC5509 25
4.7.1 A Byte is 16 Bits 25
4.7.2 64K Page Limit 25
4.7.3 Pipeline 25

4.8 Global Variables Reference 26
4.8.1 DSP Type 26
4.8.2 Interrupt Vector Table 27

4.9 Macros Reference 27
4.9.1 DebugOutByteHex 27
4.9.2 DebugOutConstString 27
4.9.3 DebugOutDwordHex 27
4.9.4 DebugOutNibbleHex 28
4.9.5 DebugOutSByteDec 28
4.9.6 DebugOutSDwordDec 29
4.9.7 DebugOutSNibbleDec 29
4.9.8 DebugOutString 29
4.9.9 DebugOutSWordDec 30
4.9.10 DebugOutUByteDec 30
4.9.11 DebugOutUDwordDec 30
4.9.12 DebugOutUNibbleDec 31
4.9.13 DebugOutUWordDec 31
4.9.14 DebugOutWordHex 31

4.10 Functions Reference 32
4.10.1 SetupDSP 32
4.10.2 C5xIntHook 32
4.10.3 C5xIntEnable 33
4.10.4 C5xIntDisable 33
4.10.5 C5xIntClear 33
4.10.6 LedOn 34
4.10.7 LedOff 34
4.10.8 LedToggle 34
4.10.9 TimerInit 34
4.10.10 TimerStart 35
4.10.11 TimerStop 35
4.10.12 FpgaLoad 35
4.10.13 FlashGetDeviceInfo 36
4.10.14 FlashEraseSector 36

4.10.15 FlashProgram 37
4.10.16 FlashRead 37
4.10.17 DebugBufmgr 37
4.10.18 DebugFlush 38
4.10.19 DebugGetc 38
4.10.20 DebugGets 39
4.10.21 DebugInit 39
4.10.22 DebugKbhit 39
4.10.23 DebugPutc 40
4.10.24 DebugPuts 40
4.10.25 DecSignedByte2Ascii 40
4.10.26 DecSignedDword2Ascii 41
4.10.27 DecSignedNibble2Ascii 41
4.10.28 DecSignedWord2Ascii 41
4.10.29 DecUnsignedByte2Ascii 42
4.10.30 DecUnsignedDword2Ascii 42
4.10.31 DecUnsignedNibble2Ascii 42
4.10.32 DecUnsignedWord2Ascii 43
4.10.33 HexByte2Ascii 43
4.10.34 HexDword2Ascii 43
4.10.35 HexNibble2Ascii 44
4.10.36 HexWord2Ascii 44

4.11 Memory Map and Register Description 44
4.11.1 Memory Map of the UC1394a-1 44
4.11.2 Internal RAM of the Processor 45
4.11.3 Flash Memory 45
4.11.4 External RAM 45
4.11.5 FPGA registers 45
4.11.6 FPGA register overview 46
4.11.7 FPGA register map 46

4.12 LLC registers 47

4.13 Interrupts 48
4.13.1 LLC interrupts 48
4.13.2 FPGA I/O pin interrupts 48
4.13.3 Streaming interrupts 48
4.13.4 External interrupts /INT3 and /INT4 48

5 DETAILED INTERFACE DESCRIPTION 49

5.1 Peripheral interface 49
5.1.1 Wait State Generator Programming 50

5.2 IEEE1394 Interface 50
5.2.1 How Data is Transferred Over IEEE1394 50
5.2.2 Plug & Play features of IEEE1394 51
5.2.3 Power Distribution Over IEEE1394 51
5.2.4 Isolation 51

5.3 Software Streaming 51

5.4 UART Interface 61

5.4.1 UART Hardware Interface 61
5.4.2 UART Programming 62

5.5 I/O Pins 65
5.5.1 I/O Pin Hardware Interface 66
5.5.2 I/O Pin Programming 66

5.6 Other Interfaces Provided by FPGA Registers 69
5.6.1 System Control Register (SYS_CTL) 69
5.6.2 Watchdog Control Register (WDG) 70
5.6.3 Version Register (VER) 71

6 HARDWARE IMPLEMENTATION GUIDELINES 73

6.1 Power Supply 73

6.2 IEEE1394 Interface 73

6.3 RS-232 Level-Converter 75

6.4 JTAG Interface 75

6.5 Unused Signals 76

6.6 Minimal Connection Example 76

7 TECHNICAL DATA 78

7.1 Signal Overview and Connector Pinout Tables 78

7.2 Individual Signal Description 81

7.3 Dimensions of the UC1394a-1 86

7.4 Environmental Conditions 86
7.4.1 Storage 86
7.4.2 Ambient Humidity 86
7.4.3 Ambient Temperature 87

7.5 Soldering Process 87

7.6 Power Requirements 88

7.7 Signal Levels and Loads 88
7.7.1 FPGA Signals 88
7.7.2 Reset Signals 89
7.7.3 DSP Signals 89
7.7.4 Analog Inputs 90
7.7.5 Other Signals 90

7.8 Peripheral Interface Timing 90
7.8.1 Timings for a 200MHz TMS320VC5509A 91
7.8.2 Timings for a 144MHz TMS320VC5509 91

7.9 Reset Timing 92

7.10 I/O Pin Timings 92

8 GLOSSARY 93

9 LITERATURE REFERENCES 94

List of Tables

Table 1: CPU address map 45
Table 2: Flash memory layout 45
Table 3: LLC and FPGA register overview 46
Table 4: LLC and FPGA register map 47
Table 5: Interrupt usage 48
Table 6: Peripheral interface signals 50
Table 7: UART connector pin assignments 62
Table 8: I/O pin assignments 66
Table 9: Pinning of the IEEE1394 connectors 73
Table 10: IEEE1394 connector part numbers 73
Table 11: Required cable connection to a host PC 75
Table 12: Pinout sorted by pins 78
Table 13: Power supply and reset signals 79
Table 14: Peripheral interface signals 79
Table 15: I/O pin signals 79
Table 16: UART interface signals 79
Table 17: DSP JTAG signals 79
Table 18: IEEE1394 signals 80
Table 19: McBSP signals 80
Table 20: ADC signals 80
Table 21: I^2C signals 80
Table 22: USB signals 80
Table 23: Power requirements 88
Table 24: Signal levels and loads for FPGA signals 89
Table 25: /RESET_IN signal levels 89
Table 26: /RESET_OUT signal levels 89
Table 27: Signal level and loads for the DSP signals 90
Table 28: Allowed input voltage range for the ADC inputs 90
Table 29: Reset timing 92

List of Figures

Figure 1: Internal block diagram of the UC1394a-1 (withDSP Master BSP) 13
Figure 2: Software development flow 19
Figure 3: Memory view of a string in character format 25
Figure 4: Memory view of a string in binary format 25
Figure 5: Peripheral interface signals 49
Figure 6: Software streaming block diagram 52
Figure 7: UART interface block diagram 61
Figure 8: UART signals 62
Figure 9: I/O pin block diagram 65
Figure 10: I/O pin configurations 66
Figure 11: 6-pin IEEE1394 connectors 74
Figure 12: 4-pin IEEE1394a connector 74
Figure 13: Pin numbering for 6-pin and 4-pin IEEE1394 connectors (top view) 74
Figure 14: Supplying the MCM from IEEE1394 74
Figure 15: Supplying power to the IEEE1394 cable 75
Figure 16: Wiring of the UART interface 75
Figure 17: Wiring of the DSP JTAG interface 76
Figure 18: Required connections 77
Figure 19: Dimensions of the UC1394a-1 (including connector pins) 86
Figure 20: Recommended PCB footprint of the UC1394a-1 86
Figure 21: Soldering temperature example 87
Figure 22: Peripheral interface read timing 90

Figure 23: Peripheral interface write timing 91
Figure 24: Peripheral interface read timing (200MHz) 91
Figure 25: Peripheral interface write timing (200MHz) 91
Figure 26: Peripheral interface read timing (144MHz) 92
Figure 27: Peripheral interface write timing (144MHz) 92

1 Preface

1.1 Document Organization

This document is organized as follows:

- Chapter 2 gives a brief overview of the whole system and its interfaces
- Chapter 3 tells how to do the very first steps
- Chapter 4 describes how to develop software for the UC1394a-1
- Chapter 5 describes each interface in detail, including the associated registers
- Chapter 6 shows how to integrate the UC1394a-1 in a customized hardware environment
- Chapter 7 lists technical data of the UC1394a-1, such as pinning and timings
- Chapter 8 explains the abbreviations that are used throughout this document
- Chapter 9 lists documents that contain further information

1.2 Documentation Overview

This chapter lists the documentation from Orsys that is shipped together with the DSP master board support package. Further documents from other vendors are listed in chapter 8 and are referenced throughout the document in square brackets.

DSP Development Kit User's Guide [15] `(DSP_DevKit_UG.pdf)`:
Shipped with the DSP development kit only. Describes the environment that the carrier board adds to the UC1394a-1 MCM along with some quick start examples.

IEEE1394 embedded API User's Guide [14] `(emb_1394_API_UG.pdf)`:
Describes the application programmer interface (API) for the IEEE1394 subsystem.

1.3 Notational Conventions

Names of registers, bit fields and single bits are written in capital letters.
Example: LLC_VERSION

Names of signals are also given in capital letters, active low signals are marked with a '/' at the beginning of the name.
Example: /RESETIN

Configuration parameters, function names, path names and file names are written in *italic* typeface.
Example: *dev_id*

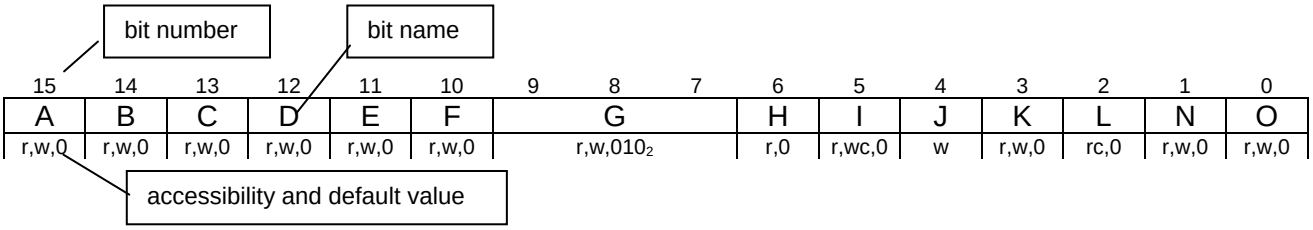
Source code examples are given in a small, fixed-width typeface.
Example: `int a = 10;`

Menus and commands from menus and submenus are enclosed in double-quotes. Example:
Create a new project using the "Create Project..." command from the "File" menu.

The members of a bit field or a group of signals are numbered starting at zero, which is the least significant bit.
Example: CFG[4:0] identifies a group of five signals, where CFG0 is the least significant bit and CFG4 is the most significant bit.

If necessary, numbers are represented with a suffix that specifies their base.
Example: $12AB_{16}$ is a hexadecimal number (base 16 = hexadecimal) and is equal to 4779_{10}.

The bit fields of a register are displayed with the most significant bit to the left. Below each bit field is a description of its read / write accessibility and its default value:


bit number
bit name
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
A B C D E F G H I J K L N O
r,w,0 r,w,0 r,w,0 r,w,0 r,w,0 r,w,0 r,w,010₂ r,0 r,wc,0 w r,w,0 rc,0 r,w,0 r,w,0
accessibility and default value

legend:

r	bit is readable
rc	this bit is cleared after a read
r,w	bit is readable and writeable, reading yields the previously written value unless otherwise specified.
w	bit is writeable, read value is undefined
wc	writing a '1' to this bit clears it
w,0	bit is write-only, reading always yields 0.
0	default value

1.4 Trademarks

TI, Code Composer, DSP/BIOS and TMS320C5000 are registered trademarks of Texas Instruments.

Microsoft® and Windows® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Hypterterminal is a trademark of Hilgraeve Inc.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

1.5 Revision History

Revision	Changes
1.0	First public release
2.0	Completely revised.
2.1	Module and footprint dimensions revised. Updated to RoHS compliant production. Implementation guidelines for IEEE1394 connectors revised. Minimum pulse width for MCM-internal reset on /RESETIN removed. UART interface: Rx/Tx FIFO mentioned. Updated to current module support library (modified function *SetupDSP*). Updated to *FlashBurn* V3.x. DSP memory map: description of reserved locations added. Different levels of IEEE1394 API support described. *FpgaLoad* in code examples updated to recommended usage.

2 System Overview

The DSP Master board support package (BSP) adds software streaming capability and a flexible peripheral interface to the UC1394a-1 multi-chip module (MCM). This creates a versatile development platform featuring low-cost and small size, ready to be used in high volume production lots.

The DSP master BSP provides the following interfaces:

- a 16-bit parallel bus interface for connecting peripherals
- Two IEEE1394a ports with 400Mbps
- UART interface
- digital I/O
- McBSP
- USB
- I^2C
- 4 ADC channels with up to 21.5kHz sampling rate
- real-time clock

The plug & play capabilities of IEEE1394 allow the UC1394a-1 to recognize other devices on the IEEE1394 network, to identify their protocol and to select a suitable partner device (such as host computers, cameras, other UC1394a-1 devices, etc.). In turn, the UC1394a-1 will be recognized as a compliant IEEE1394 device by other devices. The UC1394a-1 is fully interoperable with other devices on the bus, such as cameras, hard disks, host computers, etc.

This board support package includes:

- an API for generic high-level access to IEEE1394[1]
- routines for setting up and using the MCM's DSP
- project framework for creating own applications
- application examples

For easy start of development, the UC1394a-1 with DSP Master BSP is available in form of general purpose or specialized development kits. Please contact Orsys for further information on available development kits.

2.1 Applications

- audio
- still-image and low-bandwidth video
- high speed data transmission
- distributed data acquisition
- RS-232 over 1394
- connection of nearly any embedded system to the PC world via IEEE1394
- I/O expander for the PC via IEEE1394
- point-to-point communication in embedded systems

[1] The IEEE1394 API may be replaced by some higher-level API, depending on some development kits.

2.2 Block Diagram

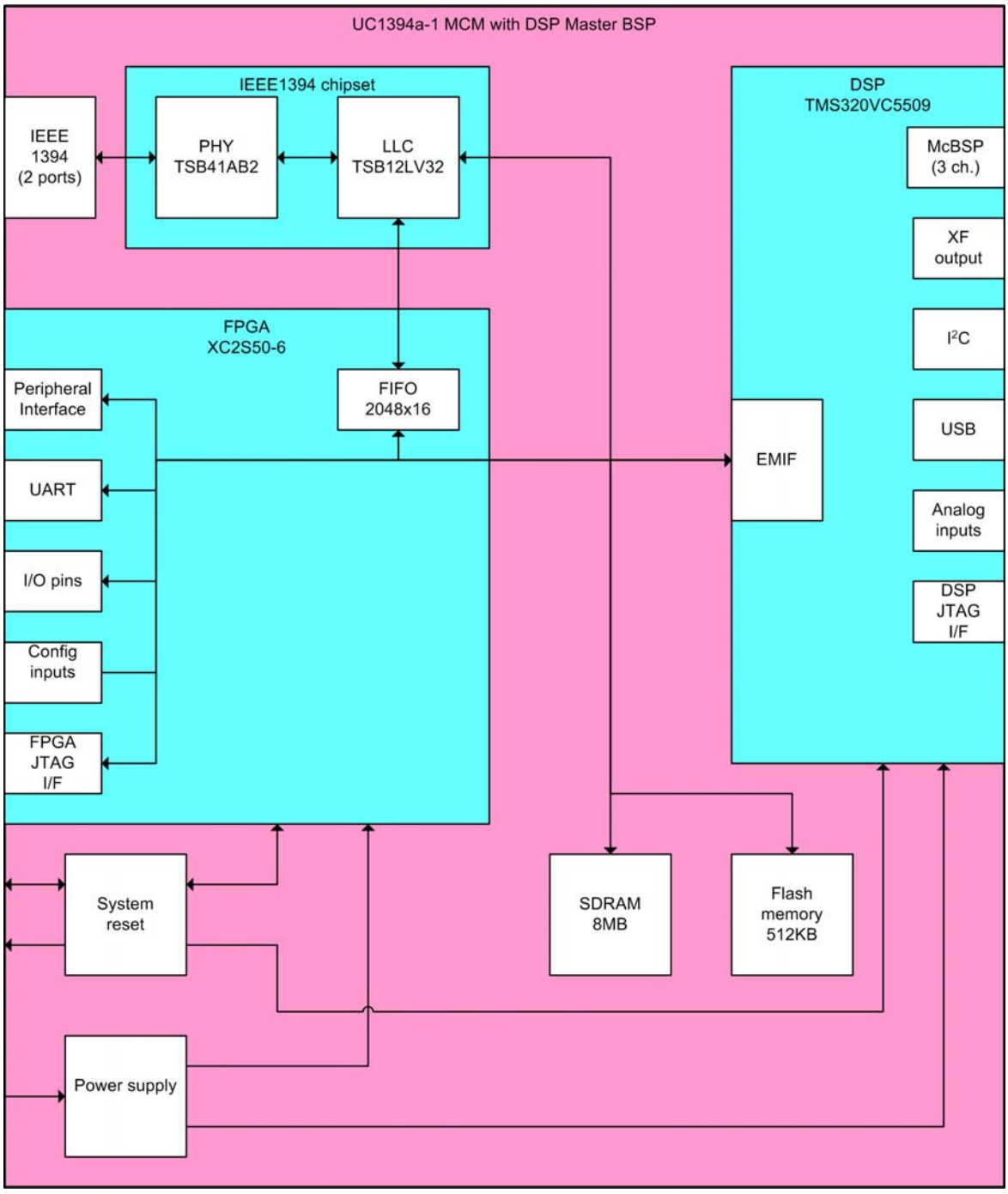


Figure 1: Internal block diagram of the UC1394a-1 (withDSP Master BSP)

2.3 DSP

The UC1394a-1 uses a 16-bit fix-point DSP. DSP type is either a TMS320VC5509 with 144MHz, or a TMS320VC5509A with 200MHz. The necessary set up of the DSP (clock, EMIF timings, interrupts) is done within the module support library which is described in chapter 4.2.1. Key features of the DSP are:

- 6.94 or 5 ns instruction cycle time with one or two instructions per cycle
- 128Kx16 on-chip memory
- on-chip peripherals, such as 2 timers, 6 DMA channels, 3 serial ports, I^2C interface, USB interface

2.4 FPGA

The UC1394a-1 uses a 50k-gate Spartan 2 FPGA. With the DSP Master BSP, the function of the FPGA is pre-defined to those features described in this document. Customized FPGA designs are possible, but require an additional development license. For details on FPGA development, please contact Orsys. Optionally, the UC1394a-1 can be equipped with a 200k-gate FPGA. Please contact Orsys for further details on availability. Depending on the DSP type, the DSP master BSP uses different FPGA code. How to check that the correct FPGA type is installed is described in chapter 5.6.3. The FPGA code is located in flash memory. It is programmed to flash memory using a dedicated programmer executable, which is included in the distribution. The FPGA is loaded on system startup as shown in chapter 4.3.

2.5 Peripheral Interface

The peripheral interface provides a straightforward connection to up to seven peripheral components such as memories, FIFOs or I/O controllers without glue-logic. A detailed description of the Peripheral interface can be found in chapter 5.1.

2.6 IEEE1394 Interface

The UC1394a-1 MCM has two 400Mbps IEEE1394 ports. The IEEE1394 interface is typically used for high-level and high-speed connections to other intelligent devices, such as host PCs or cameras. Application software accesses the IEEE1394 interface over an API, which is included in the development kits. Depending on the development kits, different levels of IEEE1394 operation are available: With the DSP Development Kit [15], customers have full access to IEEE1394 and can implement generic IEEE1394 operation with user-defined protocols. With the DCAM Frame Capture Kit, high-level access to IIDC / DCAM compliant cameras is supported.

On all kits, the following IEEE1394 features are supported:

- operate the IEEE1394 transaction layer
- bus management
- asynchronous transactions
- high-speed software data streaming

Software streaming is done over a register interface which is implemented in the FPGA. Software streaming allows transparent, low level data transfer with minimum software overhead. A detailed description of the IEEE1394 interface can be found in chapter 5.2. Software streaming is described in chapter 5.3.

2.7 UART interface

The UART interface is compatible with the standard RS-232 interfaces in personal computers. Different baud rates are supported as well as RTS / CTS handshake.

The UART interface is typically used for low-speed control and status exchange with external components or as an interface for service or debugging. A detailed description of the UART interface can be found in chapter 5.4

2.8 I/O Pins

The I/O pins provide bit-level digital I/O for controlling or sensing single digital signals. The DSP Master BSP supports

- 5 dedicated inputs
- 5 I/O pins configurable as input with interrupt capability, output or open-drain output
- 1 dedicated output pin.

A detailed description of these I/O pins can be found in chapter 5.4.1.
Further, some of the DSP on-chip interfaces, such as the McBSP interfaces can be also be configured as general purpose I/O. Details about these interfaces can be found in [5].

2.9 McBSP Interfaces

The DSP of the UC1394a-1 provides 3 McBSP ports. These interfaces are high speed serial interfaces. They support multiple channels and a lot of different operation modes, such as SPI or AC97. Using this interface, a wide range of peripherals, such as codecs, ADCs, DACs or other DSP's can be directly connected to the UC1394a-1. The McBSP interfaces can be programmed on register access level, which is described in [5]. A slightly higher level of access is provided by TI's chip support library, which is part of Code Composer Studio.

2.10 USB interface

The DSP of the UC1394a-1 has an on-chip USB interface The USB interface is implemented as a slave port and supports 12Mbps transfer rates. The USB interface can be programmed on register access level, which is described in [5]. A slightly higher level of access is provided by TI's chip support library, which is part of Code Composer Studio.

2.11 I^2C interface

The DSP of the UC1394a-1 has an on-chip I2C interface. It supports the I2C bus specification V2.1. The I2C interface can be programmed on register access level, which is described in [5]. A slightly higher level of access is provided by TI's chip support library, which is part of Code Composer Studio.

2.12 ADC

The DSP of the UC1394a-1 provides 4 ADC channels that support 10 bit sampling at up to 21.5 kHz. The ADC converter can be programmed on register access level, which is described in [5]. A slightly higher level of access is provided by TI's chip support library, which is part of Code Composer Studio.

2.13 RTC

The DSP of the UC1394a-1 has an on-chip RTC. However, the RTC is not supported by the default configuration of the MCM, because the respective MCM pins are used as I/O pins. If the RTC is required, please contact Orsys for information on RTC usage.

2.14 LED

The red LED of the UC1394a-1 MCM is available for user control and can be used for optical status display or diagnostics. How to control the LED is described in chapter 5.6.1.

2.15 System Reset

Several sources can cause a hardware reset on the UC1394a-1. A hardware reset puts the system into a well-defined state from where it can start. Sources for a hardware reset can be

- supply voltage out of bounds: This reset happens at each power on.
- Software triggered reset: Application software triggered a hardware reset over the FPGA
- Watchdog reset
- external hardware reset: An external device pulled the /RESETIN signal low. This is what is done by a reset pushbutton.

When one of these events occurs, the MCM is reset and the /RESETOUT output indicates the reset condition. Reset timing is described in chapter 7.8.2. A detailed description of a software reset can be found in chapter 5.6.1. A detailed description of the watchdog can be found in chapter 5.6.2.

2.16 DSP JTAG Interface

The DSP JTAG interface is used during development for downloading application code and debugging. During system manufacturing, the JTAG interface can be used to install the final application code (for DSP and FPGA) by programming it into the flash memory. See also chapters 4.2 and 4.3 for further information. How to connect the DSP JTAG interface to a TI emulator POD is described in chapter 6.4.

2.17 FPGA JTAG Interface

The FPGA JTAG interface is usually not used, since the FPGA code is downloaded by the DSP. It can be used optionally during FPGA development for temporary download of FPGA code. However, the loaded FPGA code is lost as soon as power is removed or the DSP is loading the FPGA.

2.18 Power Supply

The UC1394a-1 requires a single regulated supply voltage of nominal 3.3V. Please refer to chapter 7.6 for further details, such as voltage limits and current consumption.

3 Quick Start

Usually, the UC1394a-1 DSP Master BSP is shipped as part of a development kit, which includes documentation and quick start examples tailored for the kit. In contrast, this chapter describes the first steps when using the UC1394a-1 in a stand-alone environment. If you purchased UC1394a-1 DSP Master BSP as part of a kit, such as the DSP Development Kit, then please refer to the kit's documentation.

- Required items:
  - UC1394a-1 integrated into your hardware environment as described in chapter 6.
  - a JTAG emulator
  - a suitable power supply
  - a development PC with Code Composer Studio installed
  - optional: a terminal program, such as Hyperterminal and a RS-232 cable
- connect the JTAG emulator and the RS-2332 cable (optional) to the system
- power on the system
- start Code Composer Studio
- select the "Load GEL..." command from the "File" menu
- locate *uc1394a-1_master.gel* from the *GEL* folder on the distribution media and open it
- select the "Initialization"->"CPU_reset_and_init_144MHz" command from the "GEL" menu
- select the "Load Program..." command from the "File" menu
- locate one of the application examples from the examples folder on the distribution media and open it (e.g. *toggle_led.out*)
- select the "Run" command from the "Debug" menu

4 Programming the UC1394a-1

This chapter describes the software programming interfaces of the UC1394a-1. It is intended for programmers who want develop their own, customized software to run on the UC1394a-1.
The UC1394a-1 provides the following programming interfaces:

- IEEE1394 API (generic or protocol-specific)
- register-level programming of the (BSP-specific) FPGA peripherals
- on-chip peripherals of the TMS320VC5509

Programming the FPGA peripherals is described in chapter 5. Programming the on-chip DSP peripherals is supported on register level by the provided header files or by the chip support library, which is part of Code Composer Studio.

4.1 Required Tools

- TI Code Composer Studio V3.x
- FlashBurnDSK utility (included on the distribution media)
- JTAG Emulator for program download, such as TI XDS510 or other products

4.2 Software Development Flow

User defined software can be written as C-source code. The source code modules are compiled by the C-compiler. The resulting object files must be linked with at least the runtime library for the TMS320VC5509 (`rts55x.lib`). Usually, one or more object libraries are added during the linker process, such as the IEEE1394 API libraries and the module support library. The output of the linker is an executable file, which can be downloaded to the UC1394a-1 over the JTAG interface using an emulator. To store the user application permanently in flash memory, the .out file must be converted to a boot data stream by the hex conversion tool. To program this boot data stream, the *FlashBurnDSK* utility must be started. The *FlashBurnDSK* utility

- loads the Target Component executable (*FBTCOrsysUC1394a-1.out*) to the UC1394a-1 and starts it
- sends the boot data stream to *FBTCOrsysUC1394a-1*, which in turn programs it to the flash memory.

Further details on flash programming can be found in chapter 4.3.
This development flow is shown in the picture below. The distribution media contains some project examples which perform this development flow.

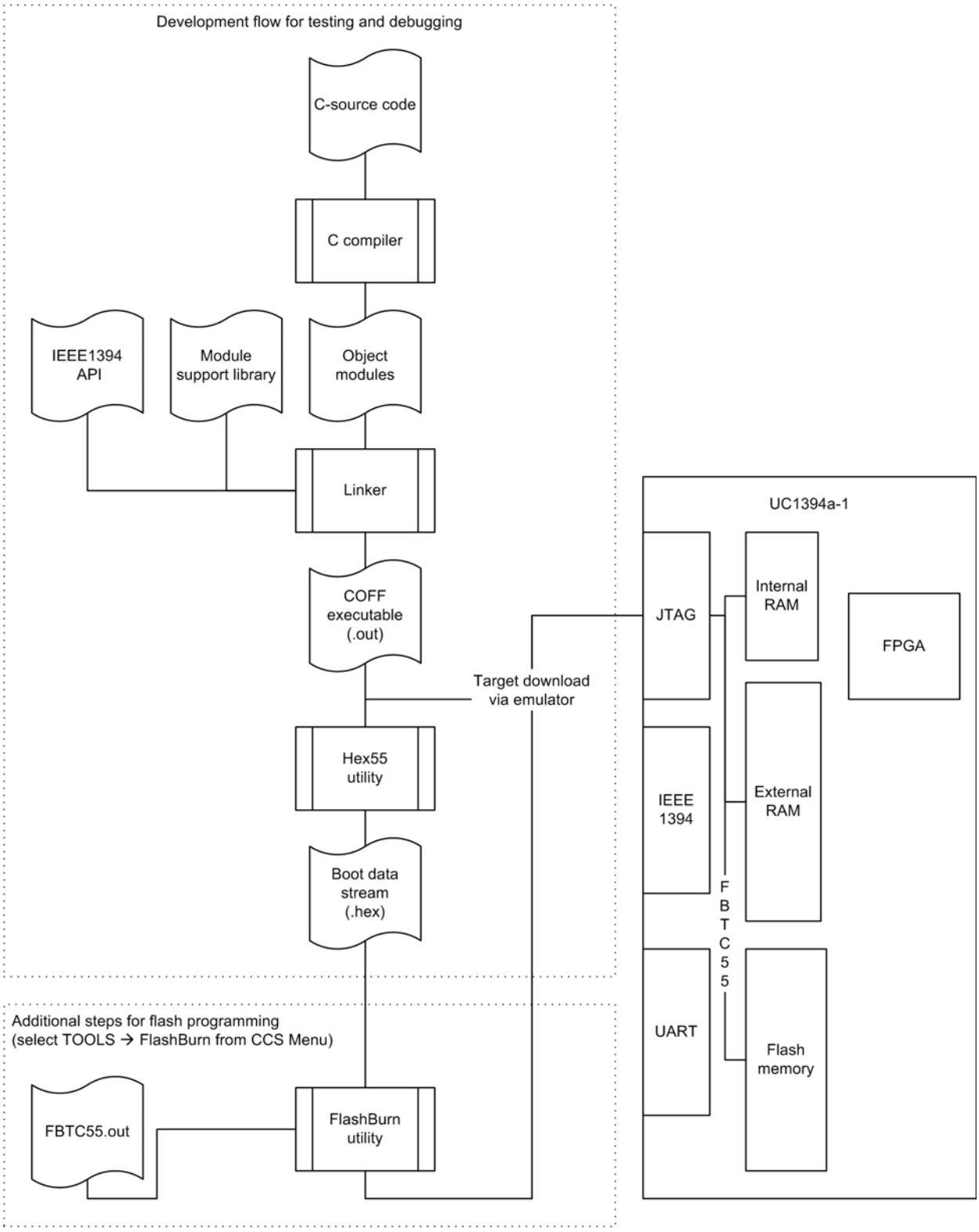


Figure 2: Software development flow

4.2.1 Module Support Library

The module support library is a collection of functions that are commonly used when programming the UC1394a-1 MCM. The code in this library is usually not changed by the user. During software development, this library is simply added to the project. However, the source code is provided for reference and for cases where a customization is necessary. A CCS project for creating the

module support library is also provided. Please note: after compiling the library sources, the compiled libraries reside in the respective output directories (*Debug* and/or *Release*). For using them with the example projects, they must be copied to the *lib\Release* and *lib\Debug* directories.
The module support library contains the following modules:

module	contents
misc.c	initialization and module support functions
fpga_load.c	FPGA loader
Flash29lv400.c	flash programming routines
debug.c	simple, buffered I/O over the UART interface
hexutil.c	binary to hexadecimal ASCII conversion
decutil.c	binary to decimal ASCII conversion

Below is a brief description of each module. The functions of module are explained in chapter 4.10.

4.2.1.1 misc.c

This module defines initialization and utility functions for the MCM, such as interrupt control.

4.2.1.2 fpga_load.c

This module contains a loader for FPGA code. The FPGA must be loaded using this loader at system startup. FPGA resources are only available after loading.

4.2.1.3 flash29lv400.c

Contains flash programming routines. It is recommended that application software does not modify the flash memory. Instead, the provided methods for accessing the flash memory should be used (*FlashBurnDSK* utility for application code programming (see chapter 4.3) or FPGA flasher executable for updating / programming FPGA code).

4.2.1.4 debug.c

This module contains a simple system for buffered character I/O over the UART interface. Typically, the functions of this module are not used directly, but over associated macros (see chapter 4.9). *debug.c* can be used as an alternative for the *stdio* functions (e.g. *printf*), especially, when small code size is required or no emulator is available.

4.2.1.5 hexutil.c

Utility functions that convert binary values to hexadecimal ASCII.

4.2.1.6 decutil.c

Utility functions that convert binary values to decimal ASCII.

4.3 Startup Procedure

After power-up or a system reset, the DSP starts its internal boot loader. The boot loader initializes the MCM according to the information of the boot header. Then, it loads the application from flash memory into internal RAM or SDRAM and starts program execution at the specified address. This is the default startup procedure in end-system environment. During development the startup procedure looks a little bit different:
After power-up or a system reset, the DSP starts its internal boot loader. The boot loader tries to load an application from flash, which may succeed or leave the DSP in an unknown state. The user starts Code Composer Studio, loads UC1394a-1.GEL and puts the DSP to an initialized state by selecting the *CPU_reset_and_init* command from the "GEL" → "Initialization" menu. Now the user application can be loaded, executed and debugged using the emulator.

4.4 System Initialization

This section shows a typical startup procedure for an application that uses

- FPGA features,
- the IEEE 1394 API,
- buffered character I/O over the UART (using *debug.c*),

such as the *asynctst* example on the distribution media.

The following header files are required:

dsp_master_bsp.h	Defines the FPGA registers and includes basic hardware definitions of the UC1394a-1.
misc.h	Contains function prototypes for the main part of the module support library, such as initialization and interrupt management.
fpga_load.h	functions for loading the FPGA.
debug.h	Defines a simple, buffered character I/O interface using the UART interface. Suitable for debugging output as well as general character I/O.
sbicfg.h	Contains various settings that were used to compile the IEEE1394 API. Required for definition of the LLC interrupt resources.
sbiapi.h	User interface of the 1394 API.
PacAsync.h	This file defines the speed codes used for IEEE1394 transactions.

In *main()*, the DSP is set up (clock, EMIF, interrupts) by calling *SetupDSP()*.Please note that no EMIF or clock initialization is specified, because this would corrupt SDRAM contents.

```
//initialize the system (clock, EMIF, interrupts, etc.)
SetupDSP(eSameSpeed);
```

The interrupt handler of the IEEE1394 API is installed using *C5xIntHook()* from the module support library.

```
//install 1394 API's interrupt handler (INT_LLC is defined in sbicfg.h)
C5xIntHook (INT_LLC, LynxHALNodeISR);
```

If other user interrupts are required they must be installed now. Below is an example for installing a receive interrupt handler (*McBsp0_Rx_ISR()*) for McBSP Interface 0:

```
C5xIntHook (C55x_RINT0, McBsp0_Rx_ISR);
C5xIntEnable (C55x_RINT0);
```

Then, the FPGA is loaded from flash memory and the FPGA version register is checked for the correct version. Please refer to chapter 5.6.3 for a list of suitable FPGA versions.

```
// load FPGA from Flash
if (FpgaLoad ((INT32U*)UC1394A_FLASH_FPGA_CODE_BASE,
              UC1394A_FLASH_FPGA_CODE_LENGTH) != FPGA_SUCCESS)
{
    while (1); // stop (we do not have a LED as indicator yet)
}

//check for correct FPGA version
usFpgaVersion = (UC1394A_VERSION & UC1394A_VERSION_VER_MASK) >> 8;
if (bIs5509A)
{
    if (usFpgaVersion != 0x07)
        while(1); //stop (we probably do not have a LED as indicator yet)
}
else
{
    if (usFpgaVersion != 0x03)
        while(1); //stop (we probably do not have a LED as indicator yet)
}
```

After successfully loading the FPGA, the red LED of the MCM is switched off.

```
//switch off red LED (which is on by default after FPGA load)
LedOff();
```

Now, the FPGA resources are available and the debugging interface (and also the UART interface) can be initialized.

```
    //initialize debugging output (this will also
    //initialize the UART interface)
    DebugInit();
```

Next part is the initialization of the 1394 API. Device information, configuration ROM location and some essential callbacks are passed to the API's initialization function. Please note that this step and subsequent steps may look different, depending on the actual development kit.

```
    /*
    ** Get and show device information
    */
    sInitInfo.ulSerialNumberLow  = UC1394A_FLASH_SERIAL_NUMBER;
    sInitInfo.ucSerialNumberHigh = UC1394_CHIP_ID_HIGH;
    DebugOutConstString("Node vendor ID, chipID high and serial number: 00B02A ");
    DebugOutByteHex(sInitInfo.ucSerialNumberHigh);
    DebugOutConstString(" ");
    DebugOutDwordHex(sInitInfo.ulSerialNumberLow);
    DebugOutConstString(" (");
    DebugOutUDwordDec(sInitInfo.ulSerialNumberLow);
    DebugOutConstString(")\r\n\r\npress '?' to get help page\r\n");
    DebugFlush();
    /*
    ** initialize the API
    */
    sInitInfo.pulDriverConfigROM        = ulConfigRom;
//  sInitInfo.ucSerialNumberHigh        = (already initialized)
//  sInitInfo.ulSerialNumberLow         = (already initialized)
    sInitInfo.pBusResetCompleteCallback = appBusResetCallback;
    sInitInfo.pErrorDetectedCallback    = appErrorDetectedCallback;
    if ((iError = sbiInitialize(&sInitInfo)) != SBI_NO_ERROR)
    {
        DebugOutConstString("sbiInit: ");
        DebugOutWordHex(iError);
        DebugOutConstString("\r\n");
        DebugFlush();
        LedOn();
        exit(-1);
    }
```

Now the API is initialized. IEEE1394 processing is started by enabling LLC interrupts and causing a bus reset.

```
    sbiCauseBusReset(); /* issue bus reset to get self IDs, etc. */
    LLC_INT_ENABLE;     /* start processing of API events        */
```

After processing the bus reset, the API creates device handles for all devices that are present.

```
    /* wait for bus enumeration to complete */
    while (sbiEnumerationCompleted() == FALSE)
    {
        DebugBufmgr();
        YieldToCallQ();
        DebugOutConstString(".");
    }
    DebugOutConstString("\r\n");
    /* show, who is present on the bus */
    ListNodes();
```

For incoming transactions, an address space is allocated. This step strongly depends on the application and the protocol that is used. The *asynctst* example simply allocates a buffer for test purposes at address 0. When implementing specific protocols, such as defined by the IIDC or SBP-2 standard, the protocol standard defines, which address ranges must be allocated.

```
    /* allocate address range for incoming transactions */
    sRangeInfo.pulBuffer             = aulBuffer;
    sRangeInfo.uiBufferLengthInBytes = 4 * ASYTST_MAX_BLOCKSIZE_IN_QUADS;
    sRangeInfo.iAccessType           = ACCESS_TYPE_READ | ACCESS_TYPE_WRITE;
    sRangeInfo.usAddressStartHigh    = 0x0000;
    sRangeInfo.ulAddressStartLow     = 0x00000000;
    sRangeInfo.iNotificationOptions  = NOTIFY_AFTER_READ | NOTIFY_AFTER_WRITE;
    sRangeInfo.pNotificationCallback = appNotificationCallback;
    iError = sbiAllocateAddressRange(&sRangeInfo);
    if (iError != SBI_NO_ERROR)
    {
        DebugOutConstString("sbiAllocateAddressRange failed: ");
        DebugOutWordHex(iError);
        DebugOutConstString("\r\n");
        DebugFlush();
        return;
    }
```

After some application specific setup the main loop is entered.

```
    /* select default destination node (node 0 or 1, which ever is remote) */
    sSetup.hRemoteNode = sbiGetDeviceHandle((sInitInfo.uiThisNodesID == 0)?
                                            1 : 0);
    DebugOutConstString("press ? for help page, "
                        "<space> to start a transaction\r\n");
    DebugFlush();
    bBusReset = FALSE;

    /* main loop */
    while(1)
    {
```

4.5 Main Loop

The main loop performs the following tasks:

- Serving the callback queue of the IEEE1394 API. All IEEE1394 transaction layer activity is handled by callback functions which are processed sequentially by calling *YieldToCallQ()*.
- handling user commands which come in over the UART interface
- IEEE1394 bus reset handling

```
/* main loop */
    while(1)
    {
        char acCmd[2];

        /* process API callbacks */
        YieldToCallQ();

        acCmd[0] = '\0';
        DebugInString(1,acCmd);
        if(acCmd[0] != '\0')
        {
            switch (acCmd[0])
            {...}
        }
        if (bBusReset == TRUE)
        {
            ListNodes();
            bBusReset = FALSE;
        }
```

4.6 How to Store an Application in Flash Memory

The UC1394a-1 supports up to 384KB of flash memory for application code. This application code must be in a format suitable for the processor's boot loader. To generate such a file, the projects on the distribution CD contain a final build step that uses the hex55 utility. To program your application into the UC1394a's flash memory, you must

- Build the project as normal.
- connect the JTAG emulator to the UC1394a-1.
- Start the *FlashBurnDSK* utility.
- Select "Create a new FlashBurn Configuration" and click on "OK".
- For "Step 1", "Connection", select a suitable Code Composer connection, such as "C5509A XDS510 Emulator (CPU_1)" and click on "Connect".
- For "Step 2", "FBTC program file", locate *FBTCOrsysUC1394a-1.out* and click on "Download FBTC". The FBTC program file is typically located on the distribution media in the folder *FlashBurn*.
- For "Step 3", "File to Burn", locate the application code, such as *toggle_led.hex* within the respective project folder, such as *examples\toggle_led\Debug*.
- Leave "Conversion Cmd File" empty.
- Click on "Erase Flash" to erase the flash memory. This erases only the application area, but not the FPGA code.
- Click on "Program Flash" to store the application to flash memory.
- Close *FlashBurnDSK*. Before exiting, *FlashBurnDSK* asks to save the current settings to a file for future use.
- power-cycle the UC1394a-1 (switch off and then on again).
- Now the application is loaded from flash memory and executed.
- In case of problems, the UC1394a-1 must be initialized by CCS as follows:
  - Select "Wiew" → "Show Code composer" from the menu.
  - In CCS, select "File" → "Load Gel" from the menu, locate UC1394a-1_master.gel" and click on "Open".
  - select the "GEL" → "Initialization" → "CPU_reset_and_init_200MHz" from the menu
  - select the "File" → "Reload Program" command from the "File" menu,
  - select the "Debug" → "Run" from the menu
  - Now Code Composer Studio must display the cursor at *DoMessageProc* in the disassembly window.
  - In *FlashBurnDSK*, repeat the last step(s).

Further help can be found in the help menu of *FlashBurnDSK* or at [16].

4.7 Hints for Programming the TMS320VC5509

4.7.1 A Byte is 16 Bits

Please note, that the TMS320VC5509 can only access data in units of 16 bit. Even a character array will consist of 16 bit. The screenshots below illustrate this.

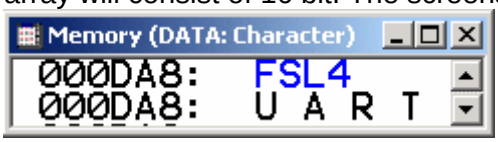


Figure 3: Memory view of a string in character format

Figure 4: Memory view of a string in binary format

Further information can be found in [4]; chapter "Memory and I/O space" and in [7]; chapter "Data types"

4.7.2 64K Page Limit

The TMS320C5000 series DSPs use a 16-bit architecture, which adds restrictions to pointer accesses. Although the DSP as well as the C-compiler support 23-bit pointers, pointer manipulation is always done modulo 64K. Below is a code example that shows how to handle arrays which cross 64K boundaries.

```
/* wrong, will stay in the lower 64K bytes */
static int array[100000];
int i;
for (i = 0; i < sizeof(array); i++)
    array[i] = 0;

/* correct: use a cast to calculate the pointer for each access */
static int array[100000];
unsigned long i;
for (i = (unsigned long)array;
     i < (unsigned long array) + sizeof(array);
     i++)
    *(unsigned long *)i = 0;
```

4.7.3 Pipeline

Accesses to memory may take some clock cycles until they are completed, since the execution is broken down into several pipelined steps. For memory accesses this is no problem. However, accesses to hardware registers can lead to unexpected results. One example is disabling interrupts:

```
    asm(" BSET INTM");
    asm(" NOP");
    asm(" NOP");
    asm(" NOP");
    asm(" NOP");
    asm(" NOP");
    asm(" NOP");
```

Without the NOP instruction, an interrupt can occur immediately after the BSET INTM instruction. The execution of the NOPs ensures, that all stages of the BSET INTM instruction have been performed before further code is executed. The NOPs must be inserted only if the code immediately following the BSET INTM instruction must be protected against interrupts.

Another point is access to peripherals in a write-read back fashion. Since a write access takes longer to be performed than a read access, the read may occur before the write, resulting in outdated data. When reading back data that has just been written, two NOP instructions should be inserted between write and read. Below is an example taken from the flash programming routines:

```
/* Wrong, do not use that */
    for (ulWords = 0; ulWords < ulLengthInWords; ulWords++)
    {
        // enter programming mode
        *(volatile INT16U*) (UC1394A_FLASH_BASE + 0x0555) = 0xAA;
        *(volatile INT16U*) (UC1394A_FLASH_BASE + 0x02AA) = 0x55;
        *(volatile INT16U*) (UC1394A_FLASH_BASE + 0x0555) = 0xA0;
........// write data word to flash
        *(volatile INT16U*)ulFlashAdr = *(volatile INT16U*)ulDataAdr;
........// wait until programmed
        while (*(volatile INT16U*)ulFlashAdr != *(volatile INT16U*)ulDataAdr)
            asm(" NOP");
        ulFlashAdr++;
        ulDataAdr++;
        if ((ulWords & FLASH_PRG_CALLBACK_RATIO) == 0 &&
            pCallback != NULL)
            pCallback();
    }

/* corrected code */
    for (ulWords = 0; ulWords < ulLengthInWords; ulWords++)
    {
        // enter programming mode
        *(volatile INT16U*) (UC1394A_FLASH_BASE + 0x0555) = 0xAA;
        *(volatile INT16U*) (UC1394A_FLASH_BASE + 0x02AA) = 0x55;
        *(volatile INT16U*) (UC1394A_FLASH_BASE + 0x0555) = 0xA0;
........// write data word to flash
        *(volatile INT16U*)ulFlashAdr = *(volatile INT16U*)ulDataAdr;
        // The following NOP's causes the write operation to finish before
        // the programming status is read. Otherwise, the pipeline could
        //  exchange write and read, which causes a premature abort.
        asm(" NOP");
        asm(" NOP");
........// wait until programmed
        while (*(volatile INT16U*)ulFlashAdr != *(volatile INT16U*)ulDataAdr)
            asm(" NOP");
        ulFlashAdr++;
        ulDataAdr++;
        if ((ulWords & FLASH_PRG_CALLBACK_RATIO) == 0 &&
            pCallback != NULL)
            pCallback();
    }
```

4.8 Global Variables Reference

The variables below are intended for use with application-specific use of the DSP on-chip timers. They can be used to store application-specific values for use by timer interrupt handlers. They are set up by *TimerInit()*. If not used, they can safely be ignored.

```
//global variables for timer operation
extern INT16U usSWPreload0;
extern INT16U usSWPreload1;
extern INT16U usSWPrescaler0;
extern INT16U usSWPrescaler1;
extern BOOL   bSWAutoReload0;
extern BOOL   bSWAutoReload1;
```

4.8.1 DSP Type

After calling *SetupDSP*, the variable shown below is set to TRUE if a 200MHz TMS320VC5509A DSP is detected, and set to FALSE if a 144MHz TMS320VC5509 is detected. This variable can be used to calculate software controlled timings and to decide which FPGA code is to be used.

```
//initialized afer calling SetupDSP(). Tells whether a 144MHz TMS320VC5509 or a
//200MHz TMS320VC5509A is present.
extern BOOL bIs5509A;

// interrupt vector table
extern struct
{
    void (*handler) (void);
    unsigned long dummy;
} vectab[32];
```

4.8.2 Interrupt Vector Table

The module support library maintains an interrupt vector table. This table is initialized by *SetupDSP*. User interrupt handlers can be inserted by calling *IntHook*. The table is accessible from outside of the module support library in order to better support debugging. Directly accessing the interrupt vector table should be avoided.

4.9 Macros Reference

This chapter lists the macros that are defined by the module support library. Currently, the only available macros are utility functions for *debug.c*.

4.9.1 DebugOutByteHex

Converts a 8-bit number into a string (hexadecimal) and puts the string into the debug transmit buffer. This is a macro that calls the functions *HexByte2Ascii* and *DebugPuts*.

defined in
debug.h

synopsis
```
void DebugOutByteHex(INT8U digit);
```

parameters

digit	8-bit number to convert and put to the debug interface

return value
none

4.9.2 DebugOutConstString

Puts a string into the debug transmit buffer. This is a macro that calls the function *DebugPuts*. It is intended for constant string output , such as `DebugOutConstString("Hello, world\r\n");`. Pointers to strings will fail because the length of the string is not known at compile time.

defined in
debug.h

synopsis
```
INT16U DebugOutConstString(char cString[]);
```

parameters

cString[]	output string for the debug interface

return value
number of characters actually written

4.9.3 DebugOutDwordHex

Converts a 32-bit number into a string (hexadecimal) and puts the string into the debug transmit buffer. This is a macro that calls the functions *HexLong2Ascii* and *DebugPuts*.

defined in
debug.h

synopsis
```
void DebugOutDwordHex(INT32U digit);
```

parameters
digit 32-bit number to convert and put to the debug interface

return value
none

4.9.4 DebugOutNibbleHex

Converts a 4-bit number (lower 4 bits of 8-bit number) into a string (hexadecimal) and puts the string into the debug transmit buffer. This is a macro that calls the functions *HexNibble2Ascii* and *DebugPuts*.

defined in
debug.h

synopsis
```
void DebugOutNibbleHex(INT8U digit);
```

parameters
digit 8-bit number to convert and put to the debug interface

return value
none

4.9.5 DebugOutSByteDec

Converts a signed 8-bit number into a string of 4 characters and puts the string into the debug transmit buffer. This is a macro that calls the functions *DecSignedByte2Ascii* and *DebugPuts*.

defined in
debug.h

synopsis
```
void DebugSByteDec(INT8S digit);
```

parameters
digit number to convert and put to the debug interface

return value
none

4.9.6 DebugOutSDwordDec

Converts an signed 32-bit number into a string of 11 characters and puts the string into the debug transmit buffer. This is a macro that calls the functions *DecSignedDword2Ascii* and *DebugPuts*.

defined in
debug.h

synopsis
```
void DebugOutsDwordDec(INT32S digit);
```

parameters
digit number to convert and put to the debug interface

return value
none

4.9.7 DebugOutSNibbleDec

Converts a signed 4-bit number into a string of two characters and puts the string into the debug transmit buffer. This is a macro that calls the functions *DecSignedNibble2Ascii* and *DebugPuts*.

defined in
debug.h

synopsis
```
void DebugOutSNibbleDec(INT8S digit);
```

parameters
digit number to convert and put to the debug interface

return value
none

4.9.8 DebugOutString

Puts a string into the debug transmit buffer. This is a macro that calls the function *DebugPuts*. Pointers to strings are allowed since the length of the string is determined at run time.

defined in
debug.h

synopsis
```
INT16U DebugOutString(char *pString);
```

parameters
pString pointer to output string for debug the interface

return value
number of characters actually written

4.9.9 DebugOutSWordDec

Converts a signed 16-bit number into a string of 6 characters and puts the string into the debug transmit buffer. This is a macro that calls the functions *DecSignedWord2Ascii* and *DebugPuts*.

defined in
debug.h

synopsis
```
void DebugOutSWordDec(INT16S digit);
```

parameters
digit number to convert and put to debug the interface

return value
none

4.9.10 DebugOutUByteDec

Converts an unsigned 8-bit number into a string of 3 characters and puts the string into the debug transmit buffer. This is a macro that calls the functions *DecUnsignedByte2Ascii* and *DebugPuts*.

defined in
debug.h

synopsis
```
void DebugOutUByteDec(INT8U digit);
```

parameters
digit number to convert and put to the debug interface

return value
none

4.9.11 DebugOutUDwordDec

Converts an unsigned 32-bit number into a string of 10 characters and puts the string into the debug transmit buffer. This is a macro that calls the functions *DecUnsignedDword2Ascii* and *DebugPuts*.

defined in
debug.h

synopsis
```
void DebugOutUDwordDec(INT32U digit);
```

parameters
digit number to convert and put to the debug interface

return value
none

4.9.12 DebugOutUNibbleDec

Converts an unsigned 4-bit number into a string of two characters and puts the string into the debug transmit buffer. This is a macro that calls the functions *DecUNibble2Ascii* and *DebugPuts*.

defined in
debug.h

synopsis
```
void DebugOutUNibbleDec(INT8U digit);
```

parameters
digit number to convert and put to the debug interface

return value
none

4.9.13 DebugOutUWordDec

Converts an unsigned 16-bit number into a string of 5 characters and puts the string into the debug transmit buffer. This is a macro that calls the functions *DecUnsignedWord2Ascii* and *DebugPuts*.

defined in
debug.h

synopsis
```
void DebugOutUWordDec(INT16U digit);
```

parameters
digit number to convert and put to the debug interface

return value
none

4.9.14 DebugOutWordHex

Converts a 16-bit number into a string (hexadecimal)) and puts the string into the debug transmit buffer. This is a macro that calls the functions *HexWord2Ascii* and *DebugPuts*.

defined in
debug.h

synopsis
```
void DebugOutWordHex(INT16U digit);
```

parameters
digit 16-bit number to convert and put to the debug interface

return value
none

4.10 Functions Reference

This chapter gives a brief description of the module support library functions.

4.10.1 SetupDSP

SetupDsp performs some additional initialization which is not done by the boot loader or GEL file:

- if explicitly specified: sets up processor clock and EMIF settings
- disables the on-chip ROM by setting the MPNMC bit
- sets up and initializes the interrupt vector with a dummy interrupt handler (*DefaultISR()*)
- disables and clears all maskable interrupts
- disables interrupts globally during initialization an enables them again when *SetupDSP* returns

Clock and EMIF settings shouldn't be modified with *SetupDSP*. Instead, these settings should be performed by the DSP boot loader (using a suitable command file for the hex conversion utility) or by CCS initialization (using a suitable GEL file). See also description of *eMode* below. Default initialization of the interrupt vector table causes deterministic behavior when uninitialized interrupts are triggered. Clearing the interrupts prevents unwanted interrupts on startup when re-starting an application during debugging.

defined in
misc.h

synopsis
```
void SetupDSP (DSP_MODE eMode);
```

parameters

DSP_MODE eMode	Controls if and how the DSP is to be set up regarding clock generator and EMIF. Must be one of:	
	eSameSpeed	Leaves clock and EMIF settings untouched. Should be used as default.
	e144MHz	Sets up CPU clock to 144 MHz and EMIF clock to 72 MHz, independent of the actual DSP type. Operation with a 200MHz TMS32VC5509A is not verified. Please note that clock initialization may destroy SDRAM contents.
	e200MHz	Sets up CPU clock to 200 MHz and EMIF clock to 100 MHz if the DSP is a 200MHz TMS32VC5509A. Please note that clock initialization may destroy SDRAM contents.

return value
none

4.10.2 C5xIntHook

defined in
misc.h

Description
Installs an interrupt handler for the given interrupt number in the interrupt vector table. The interrupt numbers are defined in *c5509.h*. The interrupt handler must be defined with the `interrupt` keyword.

synopsis
```
void C5xIntHook (int iIntNumber, void (*pHandler)(void));
```

parameters

int iIntNumber	number of interrupt to be inserted
pHandler	pointer to the interrupt handler

return value
none

4.10.3 C5xIntEnable

Enables the specified interrupt in the corresponding Interrupt Enable Register. The interrupt numbers are defined in *c5509.h*.

defined in
misc.h

synopsis
```
void C5xIntEnable (int iIntNumber)
```

parameters

int iIntNumber	number of interrupt to be enabled

return value
none

4.10.4 C5xIntDisable

Disables the specified interrupt in the corresponding Interrupt Enable Register. The interrupt numbers are defined in *c5509.h*.

defined in
misc.h

synopsis
```
void C5xIntDisable (int iIntNumber)
```

parameters

int iIntNumber	number of interrupt to be disabled

return value
none

4.10.5 C5xIntClear

Clears the specified interrupt in the corresponding Interrupt Flag Register if pending. The interrupt numbers are defined in *c5509.h*.

defined in
misc.h

synopsis
```
void C5xIntClear (int iIntNumber)
```

parameters

iIntNumber	number of interrupt to be cleared (see c5509.h)

return value
none

4.10.6 LedOn

Switches on the red LED of the MCM.

defined in
misc.h

synopsis
```
void LedOn(void)
```

parameters
none

return value
none

4.10.7 LedOff

Switches off the red LED of the MCM.

defined in
misc.h

synopsis
```
void LedOff(void)
```

parameters
none

return value
none

4.10.8 LedToggle

Toggles the red LED of the MCM (switches off if on and vice versa).

defined in
misc.h

synopsis
```
void LedOff(void)
```

parameters
none

return value
none

4.10.9 TimerInit

Initializes on of the DSP on-chip timers for use as a single-shot or periodic timer. single-shot or continuous operation as well as a software prescaler can be implemented by application software by using the parameters *bAutoReload* and *ulInterval* in conjunction with the global timer variables (see chapter 4.8).

defined in

misc.h

synopsis
```
VOID TimerInit (INT16U usTimerNo, BOOL bAutoReload, INT32U ulInterval)
```

parameters

usTimerNo	Specifies which timer to initialize. Allowed values: 0 or 1.
bAutoReload	Flag for application specific the interrupt handlers. Gets stored in *bSWAutoReload0* or *bSWAutoReload1*
ulInterval	32-bit Timer period. The 16 LSBs go to the timer period register. The next 4 bits initialize the timer prescaler. The remaining bits are stored in *ulSWPreload0* / *ulSWPreload1* and *ulSWPrescaler0* / *ulSWPrescaler1* for use by an application-specific software prescaler.

return value
none

4.10.10 TimerStart

Starts timer operation.

defined in
misc.h

synopsis
```
VOID TimerStart (INT16U usTimerNo)
```

parameters

usTimerNo	Specifies which timer to start. Allowed values: 0 or 1.

return value
none

4.10.11 TimerStop

Stops timer operation.

defined in
misc.h

synopsis
```
VOID TimerStop (INT16U usTimerNo)
```

parameters

usTimerNo	Specifies which timer to stop. Allowed values: 0 or 1.

return value
none

4.10.12 FpgaLoad

Loads code to the FPGA (usually from flash memory). Default address when loading the FPGA code from flash is `UC1394A_FLASH_FPGA_CODE_BASE`. The code length can be retrieved from the code's header by specifying `UC1394A_FLASH_FPGA_CODE_LENGTH`. When the FPGA code is linked to the application, the parameters must be modified accordingly. Please note: The FPGA code is usually preceded by a header. *FpgaLoad()* supports both, code with and without header. Care must be taken to specify the correct length for each variant. For a 50k gates FPGA, the code length is

69900 bytes without header and 69928 bytes with header. `UC1394A_FLASH_FPGA_CODE_LENGTH` always evaluates to the length without header.

defined in
fpga_load.h

synopsis
```
int FpgaLoad(INT32U *pBootData, INT32U ulLength);
```

parameters

pBootData	pointer to FPGA image (typically in flash memory)
ulLength	length of FPGA image in bytes

return value
`FPGA_SUCCESS` (zero) if FPGA image is loaded successfully, otherwise the FPGA has not been loaded. Possible reasons for this are:

- the flash area for FPGA code is not programmed or contains invalid data
- the FPGA Code was created for a device other than a XC2S50
- the FPGA Code was created with incorrect programming file options.

4.10.13 FlashGetDeviceInfo

Reads manufacturer and device ID from the flash and stores them in the specified locations. Default manufacturer ID is 0001_{16} (AMD). Default device ID is $22BA_{16}$ (29LV400).

defined in
flash29lv400.h

synopsis
```
void FlashGetDeviceInfo(INT16U *pManufacturer, INT16U *pDevice, void (*pCallback)(void));
```

parameters

pManufacturer	pointer to location where manufacturer ID is stored
pDevice	pointer to location where device ID is stored

return value
None

4.10.14 FlashEraseSector

Erases the specified sector. During the erase process, a user-specified callback function is executed to allow the application to continue processing or to indicate the progress of the operation. The callback is called whenever the erase status is queried. Since the flash memory is completely used for storing application and FPGA code (see chapter 4.11.3), application software should not modify the flash contents.

defined in
flash29lv400.h

synopsis
```
void FlashEraseSector(int iSector, void (*pCallback)(void));
```

parameters

iSector	Sector number to be erased. Allowed values: 0 .. 10.
pCallback	Pointer to a user callback function. Must be set to a valid user callback function or to NULL if no callback is used.

return value
FLASH_OK Operation was successful

4.10.15 FlashProgram

Programs data into a previously erased area. During programming, a user-specified callback function is executed to allow the application to continue processing or to indicate the progress of the operation. The callback is called each time when 250 words have been programmed to the flash. After programming, the programmed data is verified.

defined in
flash29lv400.h

synopsis
```
int FlashProgram(INT32U ulStartOffset, INT32U ulLengthInWords, INT16U *pusData,
                 void (* pCallback)(void));
```
parameters

ulStartOffset	Destination offset relative to start of the flash, specified in 16-bit words. Allowed values: 00000000_{16} ... 00040000_{16}, or one of UC1394A_FLASH_SA0_OFFS ... UC1394A_FLASH_SA10_OFFS.
ulLengthInWords	Number of 16-bit words to program.
pusData	Points to the source data.
pCallback	User callback function.

return value

FLASH_OK	Programming was successful.
FLASH_COMPARE_ERROR	Verification of the programmed data failed.

4.10.16 FlashRead

Reads a block of data from flash memory. Can be used to load application-specific data from flash.

defined in
flash29lv400.h

synopsis
```
void FlashRead(INT32U ulStartOffset, INT32U ulLengthInWords, INT16U *pusData);
```
parameters

ulStartOffset	Offset relative to start of the flash, specified in 16-bit words. Allowed values: 00000000_{16} ... 00040000_{16}, or one of UC1394A_FLASH_SA0_OFFS ... UC1394A_FLASH_SA10_OFFS.
ulLengthInWords	Number of 16-bit word to read
pusData	Specifies the location where the retrieved data is stored.

return value
None

4.10.17 DebugBufmgr

Writes one character form the debug transmit buffer to the debug interface if there is data in the transmit buffer and the underlying debug interface is ready to accept it.
Reads one character from the debug interface to the debug receive buffer if data is available and the buffer is not already full.
The function doesn't operate interrupt-driven, so it must be called periodically.

defined in
debug.h

synopsis
```
void DebugBufmgr(void);
```

parameters
none

return value
none

4.10.18 DebugFlush

Flushes the debug transmit buffer. The function just calls *DebugBufmgr* as long as the debug transmit buffer is not empty.

defined in
debug.h

synopsis
```
void DebugFlush(void);
```

parameters
none

return value
none

4.10.19 DebugGetc

Reads one character from the debug receive buffer.

defined in
debug.h

synopsis
```
int DebugGetc(void);
```

parameters
none

return value
character read from the debug receive buffer or *DEBUG_EOF* if buffer is empty

4.10.20 DebugGets

Gets a debug message from the debug receive buffer.
The debug receive buffer is read until:
a) a newline character (\n) is encountered
b) a carriage return character (\r) is encountered
c) a null-character ('\0') is encountered
d) *usMaxLen* - 1 characters are read
e) buffer is empty

defined in
debug.h

synopsis
```
unsigned short DebugGets(unsigned short usMaxLength, char *pDebugText);
```

parameters

usMaxlength	maximum size of the debug message with trailing '\0'
pDebugText	pointer to debug message.

return value
number of actually read characters.

4.10.21 DebugInit

Initializes the debug interface.

defined in
debug.h

synopsis
```
void DebugInit(void);
```

parameters
none

return value
none

4.10.22 DebugKbhit

Tests, whether the debug receive buffer is empty.

defined in
debug.h

synopsis
```
BOOL DebugKbhit(void);
```

parameters
none

return value

TRUE	there is at least one character in the debug receive buffer
FALSE	debug receive buffer is empty

4.10.23 DebugPutc

Puts one character into the debug transmit buffer.

defined in
debug.h

synopsis
```
INT8U DebugPutc(char c);
```

parameters

c	output character for debug interface

return value
number of characters actually written (0 or 1)

4.10.24 DebugPuts

Puts a message into the debug transmit buffer.

defined in
debug.h

synopsis
```
INT16U DebugPuts(unsigned short usLength, char *pDebugText);
```

parameters

usLength	number of characters in debug string (without trailing '\0')
pDebugText	pointer to debug message

return value
number of characters actually written

4.10.25 DecSignedByte2Ascii

Converts a signed 8 bit number into a character string in decimal ASCII representation. The Character string consists of 4 characters, starting with either <space> for positive numbers or '-' for negative numbers. The string contains leading zeros if the absolute value is less than 100.

defined in
decutil.h

synopsis
```
void DecSignedByte2Ascii(INT8S digit, char *pResult);
```

parameters

digit	number to be converted
pDebugText	pointer to storage for the converted string

return value
none

4.10.26 DecSignedDword2Ascii

Converts a signed 32 bit number into a character string in decimal ASCII representation. The Character string consists of 11 characters, starting with either <space> for positive numbers or '-' for negative numbers. The string contains leading zeros if the absolute value is less than 1000000000.

defined in
decutil.h

synopsis
```
void DecSignedDword2Ascii(INT32S digit, char *pResult);
```

parameters

digit	number to be converted
pDebugText	pointer to storage for the converted string

return value
none

4.10.27 DecSignedNibble2Ascii

Converts a signed 4 bit number into a character string in decimal ASCII representation. The Character string consists of 2 characters, starting with either <space> for positive numbers or '-' for negative numbers.

defined in
decutil.h

synopsis
```
void DecSignedNibble2Ascii(INT8U digit, char *pResult);
```

parameters

digit	number to be converted
pDebugText	pointer to storage for the converted string

return value
none

4.10.28 DecSignedWord2Ascii

Converts a signed 16 bit number into a character string in decimal ASCII representation. The Character string consists of 6 characters, starting with either <space> for positive numbers or '-' for negative numbers. The string contains leading zeros if the absolute value is less than 10000.

defined in
decutil.h

synopsis
```
void DecSignedWord2Ascii(INT16S digit, char *pResult);
```

parameters

digit	number to be converted
pDebugText	pointer to storage for the converted string

return value

none

4.10.29 DecUnsignedByte2Ascii

Converts an unsigned 8 bit number into a character string in decimal ASCII representation. The Character string consists of 3 characters and contains leading zeros if the result is less than 100.

defined in
decutil.h

synopsis
```
void DecUnsignedByte2Ascii(INT8U digit, char *pResult);
```

parameters

digit	number to be converted
pDebugText	pointer to storage for the converted string

return value
none

4.10.30 DecUnsignedDword2Ascii

Converts an unsigned 32 bit number into a character string in decimal ASCII representation. The Character string consists of 10 characters and contains a leading zero if the result is less than 1000000000.

defined in
decutil.h

synopsis
```
void DecUnsignedDword2Ascii(INT32U digit, char * pResult)
```

parameters

digit	number to be converted
pDebugText	pointer to storage for the converted string

return value
none

4.10.31 DecUnsignedNibble2Ascii

Converts an unsigned 4 bit number into a character string in decimal ASCII representation. The Character string consists of 2 characters and contains a leading zero if the result is less than 10.

defined in
decutil.h

synopsis
```
void DecUnsignedNibble2Ascii(INT8U digit, char *pResult);
```

parameters

digit	number to be converted
pDebugText	pointer to storage for the converted string

return value
none

4.10.32 DecUnsignedWord2Ascii

Converts an unsigned 16 bit number into a character string in decimal ASCII representation. The Character string consists of 5 characters and contains leading zeros if the result is less than 10000.

defined in
decutil.h

synopsis
```
void DecUnsignedWord2Ascii(INT16U digit, char *pResult);
```

parameters

digit	number to be converted
pDebugText	pointer to storage for the converted string

return value
none

4.10.33 HexByte2Ascii

Converts a 8 bit number into a string ("00".."FF").

defined in
hexutil.h

synopsis
```
void HexByte2Ascii(unsigned char ucNum, char *pResult);
```

parameters

ucNum	8 bit number to convert
pResult	Pointer to result. Must be at least 3 bytes.

return value
none

4.10.34 HexDword2Ascii

Converts a 32-bit number into a string ("00000000.."FFFFFFFF").

defined in
hexutil.h

synopsis
```
void HexDword2Ascii(unsigned long ulNum, char *pResult);
```

parameters

ulNum	32 bit number to convert
pResult	Pointer to result. Must be at least 9 bytes.

return value
none

4.10.35 HexNibble2Ascii

Converts a 4-bit number (lower 4 bits of 8 bit number) into a string ("0".."F").

defined in
hexutil.h

synopsis
```
void HexNibble2Ascii(const INT8U digit, char *pResult);
```

parameters

digit	8 bit number to convert
pResult	Pointer to result. Must be at least 2 bytes.

return value
none

4.10.36 HexWord2Ascii

Converts a 16-bit number into a string ("0000".."FFFF").

defined in
hexutil.h

synopsis
```
void HexWord2Ascii(INT16U  usNum, char *pResult);
```

parameters

usNum	16 bit number to convert
pResult	Pointer to result. Must be at least 5 bytes.

return value
none

4.11 Memory Map and Register Description

4.11.1 Memory Map of the UC1394a-1

The table below shows the data memory map of the TMS320VC5509 specific to the UC1394a-1 . The addresses are given as word addresses, so each location holds 16 bit of data. To access a memory location, use a pointer to the desired address.

C-code example:
```
*(int *)0x8000; /* accesses the first location of on-chip SARAM */
```

Please note:
Accesses to program memory are done using byte addresses. For example, address 1000_{16} in data memory space equals address 2000_{16} in program memory space. This is important to know when debugging your code.

The processor's on-chip peripherals are located within the I/O memory page and must be accessed using special instructions. For details please refer to [7]; chapter "The ioport keyword". The I/O page memory map is not shown here. Please refer to [2] for details.

Address	Address space	Usage
000000_{16} – 000061_{16}	on-chip	memory mapped registers / used by boot loader
000062_{16} – $003FFF_{16}$	on-chip	dual-access RAM
004000_{16} – $007FFF_{16}$	on-chip	dual-access RAM
008000_{16} – $01FFFF_{16}$	on-chip	single-access RAM
020000_{16} – $020FFF_{16}$	CE0	LLC and FPGA registers
021000_{16} – $1FFFFF_{16}$		reserved (repeated occurrence of LLC and FPGA registers)
200000_{16} – 400000_{16}	CE1	Flash Memory
400000_{16} – $5FFFFF_{16}$	CE2	SDRAM
600000_{16} – $7FFFFF_{16}$	CE3	SDRAM

Table 1: CPU address map

4.11.2 Internal RAM of the Processor

The internal RAM consists of 256 kB. It can be used by application software for any purpose. Usually, speed critical code and data should be placed in the internal RAM. Please note that the first locations in internal memory are reserved for memory mapped registers and for DSP boot loader usage.

4.11.3 Flash Memory

The Flash memory consists of 512 kB nonvolatile storage. It is used for storing application code and FPGA code. FPGA code can only be changed by means of a dedicated programmer executable supplied by Orsys. Application code can be changed as described in chapter 4.6. Application code is stored as a boot data stream, including a header that contains some initial register settings. The DSP boot loader loads sets up the required registers (especially EMIF registers) and loads application code into RAM.

Sector	Start address	Size	Usage
0	200000_{16}	16KB	Application code
1	202000_{16}	8KB	
2	203000_{16}	8KB	
3	204000_{16}	32KB	
4	208000_{16}	64KB	
5	210000_{16}	64KB	
6	218000_{16}	64KB	
7	220000_{16}	64KB	
8	228000_{16}	64KB	
9	230000_{16}	64KB	FPGA code
10	238000_{16}	64KB	

Table 2: Flash memory layout

4.11.4 External RAM

The external RAM consists of 8 MB SDRAM. It can be used by application software for any purpose.

4.11.5 FPGA registers

These registers provide access to the interfaces that are implemented in the FPGA, such as the UART interface or the peripheral interface. Further, the IEEE1394 link layer controller (LLC) is also accessed through the FPGA. How to use the FPGA registers is described separately for each interface in chapter 5.

4.11.6 FPGA register overview

Address range	Interface
020000_{16} – $02003F_{16}$	LLC registers
020040_{16} – $02004F_{16}$	System control (LED / watchdog control)
020050_{16} – $02005F_{16}$	UART
020080_{16} – $02009F_{16}$	I/O pins
$0200C0_{16}$ – $0200CF_{16}$	Software streaming
020100_{16} – $0201FF_{16}$	peripheral interface CS1 address space
020200_{16} – $0202FF_{16}$	peripheral interface CS2 address space
020300_{16} – $0203FF_{16}$	peripheral interface CS3 address space
020400_{16} – $0204FF_{16}$	peripheral interface CS4 address space
020500_{16} – $0205FF_{16}$	peripheral interface CS5 address space
020600_{16} – $0206FF_{16}$	peripheral interface CS6 address space
020700_{16} – $0207FF_{16}$	peripheral interface CS7 address space
021000_{16} – $02FFFF_{16}$	reserved (FPGA register set repeats each 1000_{16} words)

Table 3: LLC and FPGA register overview

4.11.7 FPGA register map

Register address	Register name	Description
020000_{16} – 020001_{16}	LLC_VERSION	LLC version
020002_{16} – 020003_{16}	LLC_DM_CONTROL	LLC datamover control
020004_{16} – 020005_{16}	LLC_CONTROL	LLC control
020006_{16} – 020007_{16}	LLC_INTERRUPT	LLC interrupt flags
020008_{16} – 020009_{16}	LLC_INTERRUPT_MASK	LLC interrupt mask
$02000A_{16}$ – $02000B_{16}$	LLC_CYCLE_TIMER	LLC cycle timer
$02000C_{16}$ – $02000D_{16}$	LLC_ISOPORT	LLC isochronous port
$02000E_{16}$ – $02000F_{16}$	LLC_MAINT_CONTROL	LLC maintenance control
020010_{16} – 020011_{16}	LLC_DIAGNOSTIC	LLC diags
020012_{16} – 020013_{16}	LLC_PHY_ACCESS	LLC phy access
020014_{16} – 020015_{16}	(reserved)	(reserved)
020016_{16} – 020017_{16}	(reserved)	(reserved)
020018_{16} – 020019_{16}	LLC_FIFO_STATUS	LLC FIFO status
$02001A_{16}$ – $02001B_{16}$	LLC_BUS_RESET	LLC busreset
$02001C_{16}$ – $02001D_{16}$	LLC_HEADER0	LLC header0
$02001E_{16}$ – $02001F_{16}$	LLC_HEADER1	LLC header1
020020_{16} – 020021_{16}	LLC_HEADER2	LLC header2
020022_{16} – 020023_{16}	LLC_HEADER3	LLC header3
020024_{16} – 020025_{16}	LLC_TRAILER	LLC trailer
020026_{16} – 020027_{16}	LLC_ASYNRETRY	LLC asynchronous retry
020028_{16} – 020029_{16}	LLC_ATF_FIRST	LLC ATF first
$02002A_{16}$ – $02002B_{16}$	LLC_ATF_CONT	LLC ATF cont
$02002C_{16}$ – $02002D_{16}$	LLC_ATF_CONTUP	LLC ATF contup
$02002E_{16}$ – $02002F_{16}$	LLC_ATF_BURST	LLC ATF burst
020030_{16} – 020031_{16}	LLC_GRF_DATA	LLC GRF data
020032_{16} – 020033_{16}	(reserved)	(reserved)
020034_{16} – 020035_{16}	LLC_ATF_FIRSTUP	LLC ATF firstup
020036_{16} – 020037_{16}	(reserved)	(reserved)
020038_{16} – 020039_{16}	(reserved)	(reserved)
$02003A_{16}$ – $02003B_{16}$	(reserved)	(reserved)
$02003C_{16}$ – $02003D_{16}$	(reserved)	(reserved)

Register address	Register name	Description
$02003E_{16}$ – $02003F_{16}$	(reserved)	(reserved)
020040_{16}	SYS_CTL	System control register
020041_{16}	WDG_CTL	Watchdog control
020042_{16}	CFG	Configuration inputs (readonly)
020043_{16}	VERSION	FPGA version register (readonly)
020044_{16}	WSG	Wait state generator
020045_{16} – $02004F_{16}$	(reserved)	reserved
020050_{16}	UART_DATA	UART transmit and receive register
020051_{16}	UART_CTL	UART control and status register
020052_{16} – $02005F_{16}$	(reserved)	(reserved)
020080_{16}	PCR0	I/O pin 0 control register
020081_{16}	PCR1	I/O pin 1 control register
020082_{16}	PCR2	I/O pin 2 control register
020083_{16}	PCR3	I/O pin 3 control register
020084_{16}	PCR4	I/O pin 4 control register
020085_{6} – $02009F_{16}$	(reserved)	(reserved)
$0200C0_{16}$	STR_CTRL	Streaming control register
$0200C1_{16}$	(reserved)	(reserved)
$0200C2_{16}$	STR_HEADER1	Streaming header register 1
$0200C3_{16}$	STR_HEADER2	Streaming header register 2
$0200C4_{16}$	(reserved)	(reserved)
$0200C5_{16}$	STR_FRMSZ	Streaming port frame size
$0200C6_{16}$	STR_FIFO_AFL	Streaming FIFO almost full level register
$0200C7_{16}$	STR_FIFO_AEL	Streaming FIFO almost empty level register
$0200C8_{16}$	STR_INTM	Streaming interrupt mask register
$0200C9_{16}$	STR_FLAG	Streaming flag register
$0200C3_{16}$ - 0200CD	(reserved)	(reserved)
$0200CE_{16}$	STR_DATA STR_DATA_L	Streaming data (32 bit) Streaming data (16 bit, LSB)
$0200CF_{16}$	STR_DATA_H	Streaming data (16 bit, MSB)
$0200D0_{16}$ – $0200FF_{16}$	(reserved)	reserved
020100_{16} – $0201FF_{16}$		peripheral interface CS1
020200_{16} – $0202FF_{16}$		peripheral interface CS2
020300_{16} – $0203FF_{16}$		peripheral interface CS3
020400_{16} – $0204FF_{16}$		peripheral interface CS4
020500_{16} – $0205FF_{16}$		peripheral interface CS5
020600_{16} – $0206FF_{16}$		peripheral interface CS6
020700_{16} – $0207FF_{16}$		peripheral interface CS7
020800_{16} – $1FFFFF_{16}$	(reserved)	reserved (FPGA register set repeats each 1000_{16} words)

Table 4: LLC and FPGA register map

4.12 LLC registers

Accesses to the LLC are managed by the FPGA. Therefore, the FPGA must be correctly loaded before the LLC can be accessed.
The LLC register set is usually handled by the IEEE1394 API. In most cases, there is no need to program LLC registers directly from application level.
The LLC registers are 32-bit wide, so each LLC register occupies two 16-bit locations. The most significant word (MSW) is located at the lower address and the least significant word (LSW)

at the higher address. To access a register, either 16-bit accesses or 32-bit accesses can be used. Example:

`*(int *)0x020000;` reads the MSW of the LLC version register and should yield 7115_{16}
`*(int *)0x020001;` reads the LSW of the LLC version register and should yield $38A0_{16}$.
`*(long *)0x020000` reads the complete version register and should yield $711538A0_{16}$

Please note: when writing LLC registers, the MSW must be written first. Therefore, 32-bit accesses may only occur at even addresses (as shown in the example above). Further, the GRF register must be accessed by 32-bit accesses only.

For a detailed description of LLC registers, please refer to the LLC's data manual [13].

Address:
020000_{16} – $02003F_{16}$

Please note: the LLC registers are already defined in *llc.h*, which is automatically included, when you include the BSP header (*dsp_master_bsp.h*):

```
#include "dsp_master_bsp.h" /* board support package definitions */

...
printf("LLC version register: %8.8lX\n", LLC_version);
```

4.13 Interrupts

The TMS320VC5509 has five interrupt lines: INT0 .. INT4
They are used as described below:

Interrupt line	Usage
INT0	LLC interrupts
INT1	I/O pin interrupts
INT2	Software streaming interrupts
INT3	External interrupt
INT4	External interrupt

Table 5: Interrupt usage

4.13.1 LLC interrupts

INT0 is used by the LLC. LLC interrupts must be enabled during the IEEE1394 API initialization as shown in chapter 4.3.
The LLC interrupts are then handled by API software.

4.13.2 FPGA I/O pin interrupts

All FPGA I/O pins share the same interrupt. In order to identify the interrupt source, the I/O pin control registers must be checked. In order to generate a new interrupt, all pending interrupt sources must be serviced. Programming details about I/O pin handling can be found in chapter 5.5.2.

4.13.3 Streaming interrupts

Streaming interrupts are controlled by the STR_INTM and STR_FLAG registers. Please refer to chapter 5.3.1.1.7 and 5.3.1.1.8 for details.

4.13.4 External interrupts /INT3 and /INT4.

The external MCM interrupt inputs /INT3 and /INT4 are connected to the TMS320VC5509 interrupt lines INT3 and INT4 through the FPGA. The external interrupts are falling edge triggered, just as all DSP interrupts. Connector locations of these interrupts can be found in chapter 6.

5 Detailed Interface Description

This section describes the interfaces provided by the DSP Master BSP in detail. On chip interfaces of the DSP are described by the respective documentation from TI, which is listed throughout chapter 2 for each on-chip interface.

5.1 Peripheral interface

The peripheral interface is based on the asynchronous memory interface of the DSP. The components are selected using chip select signals /IOCS[7:0]. Within each chip select up to 256 different locations are addressable by IOADDR[7:0]. The control signals are similar to other commonly used bus interfaces of DSPs or microcontrollers. They are provided as a direction-select / strobe pair (/IOSTRB, IOR/W) as well as separate strobe signals for each direction (/IOWR, /IORD). Data is transferred on IODATA[15:0] with simple, low-level bus cycles that don't use higher-lever protocols. The timing of bus cycles is defined by the EMIF settings of the DSP and should be left as set up in the project examples because the peripheral interface shares its EMIF settings with the FPGA registers. However, the bus cycles can be extended by the MCM's FPGA. A programmable wait state generator allows to adapt the peripheral interface to the speed of the connected components[2]. Very slow peripherals can further extend the bus cycle by de-asserting the IORDY signal. Two interrupt inputs (/INT[4:3]) allow interrupt driven data transfers.
The peripheral address space is mapped into the DSP address space and can be accessed with simple C-statements:

```
#include "dsp_master_bsp.h" /* board support package definitions */

/* initialization: set up for 5 wait states */
UC1394A_WSG = 5;

/* write to address 1 in chip select space 1 */
*( INT16U *)(UC1394A_CS1_BASE + 1) = 0;
```

Timing diagrams of the peripheral interface are shown in chapter 7.8. A detailed description of the peripheral interface's signals can be found in chapter 7.2.

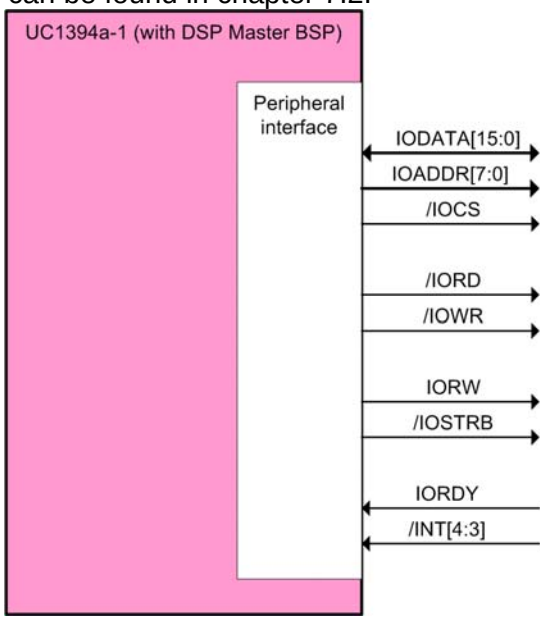


Figure 5: Peripheral interface signals

Signal	Function	direction

[2] The wait state generator is only available in 200MHz versions of the UC1394a-1 (used in current production). 144MHz Versions did not have this feature and need to modify the EMIF timings if necessary.

IODATA[15:0]	data bus	bi-directional
IOADDR[7:0]	address bus	output
/IOCS[7:0]	chip select lines	output
/IORD	read strobe (use together with /IOWR)	output
/IOWR	write strobe (use together with /IORD)	output
IOR/W	direction select (use together with /IOSTRB)	output
/IOSTRB	data strobe (use together with IOR/W)	output
IORDY	ready input for very slow peripherals	input
/INT[4:3]	interrupt inputs (routed to the respective DSP interrupts)	input

Table 6: Peripheral interface signals

5.1.1 Wait State Generator Programming

5.1.1.1 Wait State Control Register (WSG)

Address:
020044_{16}

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED												WS			
r,000000000000												r,w,0000			

WS
Number of wait states to insert for peripheral accesses. This default setting of bit field is 0. WS is encoded as follows:

WS	wait states	minimum /IOSTRB width[3]	
		in CPU clocks	in ns (at 200MHZ CPU clock)
0	none	11	55
1	1	12	65
2	2	13	75
3	3	14	85
....	...	...	...
14	14	39	195
15	15	41	205

5.2 IEEE1394 Interface

5.2.1 How Data is Transferred Over IEEE1394

Two kind of transfers are provided: isochronous transfers and asynchronous transfers. Isochronous transfers occur at intervals of 125μs at a guaranteed bandwidth. This transfer method is an excellent solution for transferring data. Isochronous transfers are multicast transfers which are identified by a channel, so there is always one talker and one or more listeners. The transfer is typically done without any software overhead. Error detection is done at the receiver side.
Isochronous transfers are well suited for

- large amounts of data
- data distribution to several devices

[3] The /IOSTRB width can vary by 1 CPU clock because the FPGA and the peripheral interface work with half of the CPU clock, which imposes one CPU clock uncertainty. The actual timing may be extended by one CPU clock (5ns at 200MHz).

- data that occurs in regular intervals

Asynchronous transfers can occur at any time (provided that the bus is free). They are point to point transfers, so the originator of the transfer must know who to talk to. An asynchronous transfer consists of a request that is sent to the destination device, and a response that the destination device sends back. This enables error checking at the sender. Asynchronous transfers are well suited for

- data that occurs randomly (e.g. control and status information)
- transfers where the originator of the transfer must be informed about the status of each single transfer

There is one more transfer method, asynchronous streaming, which is similar to isochronous transfers. Asynchronous streaming uses the same transfer elements as an isochronous transfer, but has no guaranteed bandwidth and the transfer may occur anytime, provided that the bus is free. At the receiver side it makes no difference whether isochronous or asynchronous streaming is used. This kind of transfer is well suited for situations where a minimum latency is required.

5.2.2 Plug & Play features of IEEE1394

When devices are connected to or disconnected from the IEEE1394 network, node ID's are automatically assigned for the connected devices. This is done by the chipset without any software intervention. Independent of the node ID, most devices provide some more information about themselves. There is an area within the IEEE1394 address space that is called configuration ROM. The configuration ROM holds information about

- the manufacturer of the device
- device serial number
- software interface of the device

The serial number together with the manufacturer form a world wide unique 64 bit ID. Using this 64 bit ID, the device can be identified independently of the network topology or the currently assigned node ID.

The next higher level of identification is the protocol level. By default, the UC1394a-1 when equipped with the DSP Master BSP identifies itself as a device running a generic protocol specified by Orsys. This protocol can be used on a host PC to load appropriate device drivers. customized protocol identification is available at Orsys on request.

5.2.3 Power Distribution Over IEEE1394

The IEEE1394 standard defines a 6-wire cable, that allows to supply devices over the cable. This is often used for digital cameras, for example. To operate the UC1394a-1 powered from the IEEE1394 cable, an external voltage regulator is required. The IEEE1394 standard allows up to 10W power consumption for a device.

5.2.4 Isolation

The IEEE1394 interface of the UC1394a-1 is directly connected to the remaining circuit. There is no galvanic isolation between the IEEE1394 cable and the local power supply. In a custom hardware design, the VG pin of the 1394 connector must be connected to the GND pins of the UC1394a-1.

5.3 Software Streaming

Software streaming allows to transfer large amount of data between the DSP and IEEE1394 with minimal overhead. Data transfers are buffered by a FIFO, so that the DSP can operate independent of the IEEE1394 timing. Streaming transfers are unidirectional and must be set up with the IEEE1394 API as well as the streaming registers in the FPGA. The maximum transfer rate for streaming is 32,768,000 byte/s. Software streaming uses isochronous streaming. Isochronous streaming is explained in chapter 5.2.1. The DSP Master BSP allows to

- transmit synchronization information in the data stream: see description of the STR_FRMSZ register

- synchronize receive operation to the incoming data stream: See description of the RXSYNC bit in the STR_CTL register.

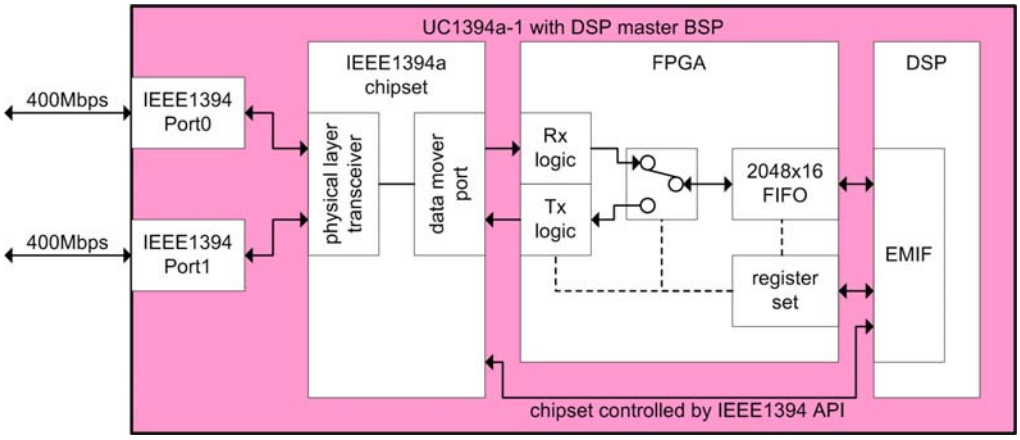


Figure 6: Software streaming block diagram

5.3.1.1 Streaming Programming

Programming the streaming port is usually a straightforward process. First, streaming is set up by programming

- FPGA registers,
- the IEEE1394 chipset (over IEEE1394 API calls *sbIIsoListen* / *sbIIsoTalk*),
- DSP interrupts (if required),
- the DMA controller of the DSP (if required).

Then, streaming data is transferred in the configured direction by accessing the STR_DATA register. Transfers are triggered by the FIFO fill level, which affects the FIFO flags in the STR_INTF register. Application software can poll these bits or use interrupt driven transfers. Interrupts can be enabled in the STR_INTM register. The FIFO fill level controls the AE and AF flags of the STR_INTF register according to the trigger levels programmed to STR_AEL and STR_AFL.

Please note that setting up streaming must be done in a different order, depending on the desired transfer direction. This avoids bus contention between the FPGA and the IEEE1394 chipset. The correct sequence is listed in the description of the DIR bit in the STR_CTL register below. A safe default is to set up the FPGA to receive from the 1394 chipset and to set up the 1394 chipset for transmit. This ensures that neither side is driving data.

Programming examples for software streaming are described in the respective kit documentation, such as [15].

5.3.1.1.1 Streaming Control Register (STR_CTL)

Description:
This register controls the basic operation of the streaming port.

Address:
$0200C0_{16}$

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED								RXSYNC	RSV		RCYSTART	LOCK	EN	RST	DIR
r,00000000								r,w,0	r,00		r,0	r	r,w,0	w,0	r,w,0

Streaming direction (DIR)
This bit controls the direction of the streaming port as follows:

DIR	Direction
0	receive (from 1394 network to streaming port) (default)
1	transmit (from streaming port to 1394 network)

The FIFO of the streaming port is not cleared by a direction change. Therefore, data can be written to it and read back for test purposes.

Notes on direction change:
1.) A direction change on the streaming port always requires that the DMRX bit of the LLC DM Control register is set accordingly. This is usually done by 1394 API software (e.g. by calling *sbIIsoListen* / *sbIIsoTalk*, preceded by *sbIIsoStop*, if necessary). To avoid bus contention on the between FPGA and IEEE1394 chipset, the following sequences must be performed:

a) direction change from receive to transmit:
- First change direction of LLC (call *sbIIsoStop*, then *sbIIsoTalkt*)
- then set the DIR bit in the streaming port's control register.

b) direction change from transmit to receive:
- First change direction of the streaming port by clearing the DIR bit in the streaming port's control register.
- then change direction of LLC (call *sbIIsoStop*, then *sbIIsoListen*)

2.) Before the direction is changed, it must be guaranteed that no transfer between FPGA and LLC is currently active. This can be done by checking the FIFO empty condition before a direction switch, or by resetting the streaming port. In receive direction, the LLC receive operation must be disabled before a direction switch (see 1.)).

Streaming reset (RST)
This bit is only writeable. A read always returns '0'. When a '1' is written to this bit, the streaming port will be immediately reset. Resetting the streaming port
aborts all current transfers
clears the internal FIFO of the streaming port

Streaming enable (EN)
This bit can be used to control streaming operation. It must be set to enable streaming operation. Setting this bit to '0' stops streaming operation. Transmit operation between FPGA and the IEEE1394 chipset is stopped at packet boundaries, whereas receive operation between the IEEE1394 chipset and the FPGA is stopped immediately.

EN	streaming operation
0	disabled (default)
1	enabled

Clock generator lock status (LOCK)
This bit is read-only. It indicates that the FPGA internal clock (used for streaming) is stable (LOCK = 1). This bit is mainly intended for diagnostic purpose and can be ignored. However, if application software accesses the streaming registers within 1µs after loading the FPGA, the LOCK bit should be polled as follows:

```
#include "dsp_master_bsp.h" /* board support package definitions */

FpgaLoad(...);
/* wait until FPGA-internal clock is stable */
while ((UC1394A_STR_CTL & UC1394A_STR_CTL_DLL_LOCK) == 0);
/* set up streaming */
UC1394A_STR_CTL = ...;
```

...

Incoming cycle start detect (RCYSTART)
This bit is read-only. If set (RCYSTART = 1), it indicates, that a cycle start packet was received from the IEEE1394 bus. Application software can poll this bit before starting streaming operation to ensure, that a cycle master is present. Without a cycle master, isochronous streaming can't work and no packets are transmitted. Please note that this bit is cleared after reading. Therefore, to get the most recent state of this bit, it should be read twice. Programming example:

```
#include "dsp_master_bsp.h" /* board support package definitions */
...
/* update the cycle start detect bit by a dummy read */
UC1394A_STR_CTL;
/* get the current time from the IEEE1394 chipset */
ulNow = sbiGetElapsedCycleTime(0);
/* wait with two cycles timeout (250us) for a cycle start to appear */
while(sbiGetElapsedCycleTime(ulNow) < 0x00002000)
{
    if (UC1394A_STR_CTL & UC1394A_STR_CTL_RCYSTART)
        break; /* cycle start detected */
}
if ((UC1394A_STR_CTL & UC1394A_STR_CTL_RCYSTART) == 0)
{
    /* error handling, no cycle master present */
}
```

The RCYSTART bit is usually set, because the IEEE1394 API has cycle master capability. A missing cycle start is usually an indicator for problems with bus management.

Receive synchronization (RXSYNC)
This bit can be used by application software to synchronize to the next packet that contains a matching sync bit pattern. When RXSYNC is set by application software, all incoming data is skipped until a packet with matching sync bits is received. Sync bits match if at least one bit is set in both, the SYNC bit field of incoming packet's header, and the *iSyncBits* parameter of *sbiIsoTalk*. The RXSYNC feature is used for receive operation only and has no effect for transmit. Please note that the RXSYNC feature uses a sync pattern set up by the IEEE1394 API, whereas the SYNC bit field of the STR_HDR1 register is not used for receive operation.

5.3.1.1.2 Streaming Header Register 1 (STR_HDR1)

Description:
This register specifies one half of the header for transmitted packets. In receive direction, this register is ignored. Do not program the header register of the IEEE1394 chipset directly. The value in the header register of the LLC will be overwritten by the value in the STR_HDR1 register. Default values and shift constants for this register are defined in the *dsp_master_bsp* header file. Programming example:

```
#include "dsp_master_bsp.h" /* board support package definitions */
...
    UC1394A_STR_HDR1 =
        ((UC1394A_STR_HDR1_TAG_UNFORMATTED << UC1394A_STR_HDR1_TAG_SHIFT)   |
         (1                                << UC1394A_STR_HDR1_CH_NO_SHIFT)  |
         (UC1394A_STR_HDR1_TCODE_ISOCH_STR << UC1394A_STR_HDR1_TCODE_SHIFT)  |
         (1                                << UC1394A_STR_HDR1_SYNC_SHIFT));
```

Address:
$0200C2_{16}$

Encoding:

15 14	13 12 11 10 9 8	7 6 5 4	3 2 1 0
TAG	CHANNEL_NUMBER	TCODE	SYNC
r,w,00	r,w,000000	r,w,1010	r,w,0000

SYNC
This bit field specifies the sync bit pattern. The LLC inserts this bit pattern in the header of the first packet of a frame (A frame is a contiguous sequence of isochronous packets). The default for this bit field is 0. On the UC1394a-1, the frame size is set up in the STR_FRMSZ register. For the

IEEE1394 API, the frame size must be set equal to the packet size (parameter *lQuadletsPerFrame* of *sbiIsoTalk* must be set to 4 * the value of parameter *iPacketDataSizeBytes*). Application software can use the SYNC bit field for synchronization on the receiver side and to indicate a new section of streaming data, such as a new picture frame in case of image (camera) data. See also the description of the RXSYNC bit in the STR_CTL register.

TCODE
This bit field specifies the transaction code for the packet to be sent. The default value is A16 (defined as `UC1394A_STR_HDR1_TCODE_ISOCH_STR` in *dsp_master_bsp.h*), which identifies the packet as a isochronous streaming packet. Application software should not change this value.

CHANNEL_NUMBER
The channel number specifies the isochronous channel for outgoing packets. Several data sources can perform isochronous streaming on different channels simultaneously (provided that the overall bandwidth isn't exceeded).

TAG
This bit field specifies the type of the payload data. The data type is application specific, however, the default for unformatted data is 0, which is defined as `UC1394A_STR_HDR1_TAG_UNFORMATTED` in *dsp_master_bsp.h.* This value is the default value, which shouldn't be changed by the application software.

5.3.1.1.3 Streaming Header Register 2 (STR_HDR2)

Description:
This register specifies second half of the header for transmitted packets. It specifies the payload data size for one packet. In receive direction, this register is ignored. Do not program the header register of the IEEE1394 chipset directly. The value in the header register of the LLC will be overwritten by the value in the STR_HDR2 register.

Address:
$0200C3_{16}$

Encoding:

15 14 13	12 11 10 9 8 7 6 5 4 3 2	1 0
RSV	PACKET_DATA_LENGTH	RSV
r,000	r,w,00000000001	r,00

Packet payload size in quadlets (PACKET_DATA_LENGTH)
This bit field determines the payload size of a packet, counted in quadlets. Together with the two reserved LSBs, the payload size can be specified in bytes. Default value is 1 quadlet (= 4 bytes).
Programming example:

```
#include "dsp_master_bsp.h" /* board support package definitions */
...
INT16U usPacketSizeInBytes = 0x100; //256 bytes payload (=64 quadlets)
    UC1394A_STR_HDR2 = usPacketSizeInBytes;
```

5.3.1.1.4 Streaming Frame Size (Number of Packets per Frame) (STR_FRMSZ)

Description:
This register defines the number of packets that are grouped together in a frame. It is only used for transmit direction (from MCM to IEEE1394). The header of the first packet of a frame has its SYNC bit field set to the value configured in the STR_HDR1 register. The remaining packets have a SYNC bit field of 0000_2. Writing to this register sets the number of packets per frame and resets the packet counter, so that the next packet will be treated as the start of a new frame. This register must be programmed with number of packets per frame – 1, so for a frame of 5 packets a value of

4 must be programmed to STR_FRMSZ. By default, STR_FRMSZ is set to 0, which means a frame size of 1 packet, so that the SYNC bit fields of each packet header are set according to the SYNC bit field in STR_HDR1.

Address:
$0200C5_{16}$

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STR_FRMSZ															
r,w,0000_{16}															

5.3.1.1.5 Streaming port FIFO almost full level register (STR_FIFO_AFL)

Description:
This register controls the almost full (AF) flag of the streaming port's FIFO. If the FIFO contains at least AF_LEVEL bytes, AF will become active. This register can be used to implement a flow control mechanism. Application software typically uses this register when receiving streaming data and sets it to the packet payload size. This causes the AF flag to be set whenever one complete packet is received. However, due to the limited FIFO size, it is recommended to use a smaller value for packets with 2048 or more bytes payload. Otherwise the FIFO is not emptied until it is (nearly) full and FIFO overflows may occur. Programming example:

```
#include "dsp_master_bsp.h" /* board support package definitions */

sSetup.uiIsoPacketsizeInQuads = 0x100; /* 1024 bytes per packet */
...
/* this is the default and very safe trigger method: The FIFO */
/* causes an interrupt whenever a complete packet îs in the   */
/* FIFO. This is safe, but doesn't allow full bandwidth.      */
UC1394A_STR_FIFO_AFL = 4 * sSetup.uiIsoPacketsizeInQuads;
```

Address:
$0200C6_{16}$

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED			AF_LEVEL												
r,000			r,w,$1FE0_{16}$												

Almost full level (AF_LEVEL)
This bit field determines the almost full level in bytes. Only even numbers of bytes are allowed. AF_LEVEL[0] is ignored and is always set to 0.

5.3.1.1.6 Streaming port FIFO almost empty level register (STR_AEL)

Description:
This register controls the almost empty (AE) flag of the streaming port's FIFO. If the FIFO contains AE_LEVEL or less bytes, the almost empty flag will become active. This register can be used to implement a flow control mechanism. Application software typically uses this register when transmitting streaming data and sets it to the FIFO size minus packet payload size. This causes the AE flag to be set whenever one complete packet fits into the FIFO. However, due to the limited FIFO size, it is recommended to use a smaller value for packets with 2048 or more bytes payload. Otherwise, the FIFO can't be filled in time and transmit performance will be degraded. Programming example:

```
#include "dsp_master_bsp.h" /* board support package definitions */

sSetup.uiIsoPacketsizeInQuads = 0x100; '/* 1024 bytes per packet */

/* this is the default and very safe trigger method: The FIFO     */
/* causes an interrupt whenever a complete packet fits into the   */
/* FIFO. This is safe, but doesn't allow full transmit bandwidth. */
UC1394A_STR_FIFO_AEL = (UC1394A_STR_FIFO_SIZE - 4 * sSetup.uiIsoPacketsizeInQuads);
```

Address:
$0200C7_{16}$

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED			AE_LEVEL												
r,000			r,w,$001E_{16}$												

Almost empty level (AE_LEVEL)
This bit field determines the almost empty level in bytes. Only even numbers of bytes are allowed. AE_LEVEL[0] is ignored and is always set to 0.

5.3.1.1.7 Streaming port interrupt mask register (STR_INTM)

Description:
This register controls the generation of interrupts for the streaming port. Application software can use this register to enable and disable interrupts that trigger data transfers or indicate error conditions. Please note: although the FPGA interrupts can automatically trigger DMA transfers, the transfers should always be triggered by software. The reason for this is the fact that spurious interrupts may occur during the transfer, caused by simultaneous FIFO access from the IEEE1394 chipset and the FPGA.

Address:
$0200C8_{16}$

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED									DMERR	UN	OV	F	AF	AE	E
r,000000000									r,w,0	r,w,0	r,w,0	r,w,0	r,w,0	r,w,0	r,w,0

Empty interrupt enable (E)
If this bit is set to 1, interrupts are generated while the empty flag is set.

Almost empty interrupt enable (AE)
If this bit is set to 1, interrupts are generated while the almost empty flag is set. Application software typically uses this bit for triggering transfers in transmit direction. To trigger transmits by the AE flag, application software must:

- program the AE level to FIFO size (4096+2 bytes) minus the transfer size in bytes:
```
UC1394a_STR_AEL = 4098 - 100; //interrupt when >= 100 bytes fit in FIFO
```
- install an interrupt handler for DSP interrupt 2:
```
C5xIntHook(C5509_INT2, IsoTxIntHandler);
```
- clear and enable DSP interrupt 2:
```
C5xIntClear(C5509_INT2);
C5xIntEnable(C5509_INT2);
```
- clear and enable FPGA interrupts through the AE flag:
```
UC1394A_STR_INTF = UC1394A_STR_INTF_AE;
UC1394A_STR_INTM = UC1394A_STR_INTM_AE;
```
- disable the interrupt after it has occurred:
```
UC1394A_STR_INTM = 0;
```
- transfer the data or start a data transfer using DMA
- clear the interrupt after the transfer:
```
UC1394A_STR_INTF = UC1394A_STR_INTF_AE;
```
- re-enable the interrupt for the next transfer:
```
UC1394A_STR_INTM = UC1394A_STR_INTM_AE;
```

Almost full interrupt enable (AF)
If this bit is set to 1, interrupts are generated while the almost full flag is set. Application software typically uses this bit for triggering transfers in receive direction. To trigger reception by the AF flag, application software must:

- program the AF level to the transfer size in bytes:
```
UC1394a_STR_AFL = 100; //interrupt when >= 100 bytes are available
```
- install an interrupt handler for DSP interrupt 2:
```
C5xIntHook(C5509_INT2, IsoRxIntHandler);
```
- clear and enable DSP interrupt 2:
```
C5xIntClear(C5509_INT2);
C5xIntEnable(C5509_INT2);
```
- clear and enable FPGA interrupts through the AE flag:
```
UC1394A_STR_INTF = UC1394A_STR_INTF_AF;
UC1394A_STR_INTM = UC1394A_STR_INTM_AF;
```
- disable the interrupt after it has occurred:
```
UC1394A_STR_INTM = 0;
```
- transfer the data or start a data transfer using DMA
- clear the interrupt after the transfer:
```
UC1394A_STR_INTF = UC1394A_STR_INTF_AF;
```
- re-enable the interrupt for the next transfer:
```
UC1394A_STR_INTM = UC1394A_STR_INTM_AF;
```

Full interrupt enable (F)
If this bit is set to 1, interrupts are generated while the full flag is set. Application software can use this bit for testing (e.g. FIFO size detect) and for detecting situations where the FIFO is about to overflow. However, usually the full condition is only polled through the STR_INTF register, but FIFO full interrupts are not used.

FIFO overflow interrupt enable (OV)
If this bit is set to 1, an interrupt is generated when data is written to a full FIFO. Application software can use this bit for error checking. However, the OV bit is usually only polled through the STR_INTF register, but overflow interrupts are not used.

FIFO underflow interrupt enable (UN)
If this bit is set to 1, an interrupt is generated when data is read from an empty FIFO. Application software can use this bit for error checking. However, the UN bit is usually only polled through the STR_INTF register, but underflow interrupts are not used.

Datamover error interrupt enable (DMERR)
If this bit is set to 1, an interrupt is generated when the LLC signals a datamover error. Application software can use this bit for error checking. However, the DMERR bit is usually only polled through the STR_INTF register, but data mover error interrupts are not used.

5.3.1.1.8 Streaming port flag register (STR_FLAG)

Description:
This register contains various flags, most of them contain information related to the streaming port's FIFO. It can be used by application software for polling the FIFO fill level and triggering transfers or for error detection. The E, AE, AF and F bits are being set permanently while the respective fill level condition is active. Therefore, if for example the AF flag is cleared while the FIFO is still in almost full condition, the AF flag is set again, and the DSP receives a new interrupt. All other flags are set by a temporary condition, which is only active for a short time.

Address:
$0200C9_{16}$

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED									DMERR	UN	OV	F	AF	AE	E
r,000000000									r,wc,0	r,wc,0	r,wc,0	r,wc,0	r,wc,0	r,wc,1	r,wc,1

Empty flag (E)
This bit will be set whenever the streaming port's FIFO is empty. It can be cleared by writing a '1' to it. If the FIFO remains empty, the empty flag will be set again after it was cleared. To get the most recent state of the FIFO,. this bit should be cleared before it is checked. Application software can use this bit for error checking. Usually, before and after a transfer, the FIFO should be empty. Programming example:

```
/* Check: FIFO must be left empty from previous operation */
UC1394A_STR_INTF = UC1394A_STR_INTF_E;
PIPELINE_DELAY;
if ((UC1394A_STR_INTF & UC1394A_STR_INTF_E) == 0)
{ ... }
```

Almost empty flag (AE)
This bit will be set whenever the streaming port's FIFO is almost empty (see chapter 5.3.1.1.6 for details). It can be cleared by writing a '1' to it. If the FIFO remains almost empty, the almost empty flag will be set again after it was cleared. To get the most recent state of the FIFO,. this bit should be cleared before it is checked (as described above for the E bit). Application software can poll this bit for triggering transfers in transmit direction (see description of the AE bit in the STR_INTM register). When DSP interrupts are used, the AE flag should be cleared after a transfer (triggered by the AE flag) has finished. This ensures that a new interrupt is generated, either immediately, or, if the FIFO is no longer almost empty, when it is almost empty again.

Almost full flag (AF)
This bit will be set whenever the streaming port's FIFO is almost full (see chapter 5.3.1.1.5 for details). It can be cleared by writing a '1' to it. If the FIFO remains almost full, the almost full flag will be set again after it was cleared. To get the most recent state of the FIFO,. this bit should be cleared before it is checked (as described above for the E bit). Application software can poll this bit for triggering transfers in receive direction (see description of the AF bit in the STR_INTM register). When DSP interrupts are used, the AF flag should be cleared after a transfer (triggered by the AF flag) has finished. This ensures that a new interrupt is generated, either immediately, or, if the FIFO is no longer almost full, when it is almost full again.

Full flag (F)
This bit will be set whenever the streaming port's FIFO is full. It can be cleared by writing a '1' to it. If the FIFO remains full, the full flag will be set again after it was cleared. Application software can use this bit for test purposes, e.g. for determining the FIFO size.

Streaming port FIFO overflow flag (OV)
This bit indicates an overflow condition, this means an attempt was made to write to the FIFO of the streaming port while the FIFO was full.
An overflow occurs, if

- in receive direction: the DSP does not read the data from STR_DATA fast enough (slower than the IEEE1394 chipset).
- in transmit direction: the DSP writes to STR_DATA faster than the IEEE1394 chipset reads the data.

The overflow flag will be set each time, a write to a full FIFO is performed. To clear this error indication, write a '1' to the OV bit. In general, overflow errors should be avoided by triggering data transfers by the fill level related interrupts / FIFO flags (AE for transmit, AF for receive). Application software should regularly poll this bit for detecting transfer errors.

Streaming port FIFO underflow flag (UN)
This bit indicates an underflow condition, this means an attempt was made to read from the FIFO of the streaming port while the FIFO was empty.
An underflow occurs, if

- in receive direction: the DSP reads more data than available (faster than the IEEE1394 chipset fills the FIFO).
- in transmit direction: the DSP writes to STR_DATA slower than the IEEE1394 chipset transmits the data.

The underflow will be set each time a read from an empty FIFO is performed. To clear this error indication, write a '1' to the UN bit. In general, underflow errors should be avoided by triggering data transfers by the fill level related interrupts / FIFO flags (AE for transmit, AF for receive). Application software should regularly poll this bit for detecting transfer errors.

Datamover error flag (DMERR)
This bit is set whenever the LLC signals an error on the datamover port. The following conditions cause this bit to be set:
invalid datamover configuration of the LLC (doesn't happen if the IEEE1394 API is used for LLC programming)
header error in receive direction
CRC error in receive direction
Application software should regularly poll this bit for detecting transfer errors.

5.3.1.1.9 Streaming FIFO Fill Level Register (STR_FIFO_LEVEL)

This register contains the number of 16-bit words that are currently present in the FIFO. Application software can use this bit for test purposes. This register is read-only.

Address:
0200CD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED				FIFO_LEVEL											
r, 0000				r, 000000000000											

5.3.1.1.10 Streaming Data Registers(STR_DATA, STR_DATA_L, STR_DATA_H)

Description:

These registers are used to transfer streaming data between the DSP and the streaming FIFO in the FPGA. Application software can use either software controlled transfers or use DMA transfers. In both cases it is recommended to access the data with 32-bit accesses, using STR_DATA. This allows faster accesses than using STR_DATA_L and / or STR_DATA_H separately. However, if 16-bit transfers must be used for some reason, STR_DATA_L or STR_DATA_H can be used in any sequence, since both registers are identical. In receive direction, STR_DATA is read only. Please note that for 16-bit accesses, the most significant word is always transferred first. The FIFO can be read back for test purposes in either direction. However, writing to the FIFO is only allowed in transmit direction (DIR bit in STR_CTL = 1). Programming example:

```
#include "dsp_master_bsp.h" /* board support package definitions */
...
/* write one quadlet to the stremaing port */
    UC1394A_STR_DATA = 0x123345678;
/* write some 16-bit words to the FIFO */
    UC1394A_STR_DATA_L = 0x0001; /* MSW 2nd quadlet */
    UC1394A_STR_DATA_L = 0x0002; /* LSW of 2nd quadlet */
    UC1394A_STR_DATA_H = 0x0003; /* MSW of 3rd quadlet*/
    UC1394A_STR_DATA_H = 0x0004; /* LSW of 3rd quadlet */
    UC1394A_STR_DATA_L = 0x0005; /* MSW of 4th quadlet */
    UC1394A_STR_DATA_H = 0x0006; /* LSW of 4th quadlet */
```

Address:

0200CE..0200CF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIFO_DATA															
r, w, x															

5.4 UART Interface

The UC1394a-1 MCM has an UART interface that can be used for standard asynchronous communication. Different baud rates are supported as well as RTS/CTS handshake. The signals of the UART interface operate with LVTTL logic levels, therefore an external level converter is required for usage as an RS-232 interface (see chapter 6.3 for an example). Incoming and outgoing characters are buffered by a 16-character FIFO. The UART interface is typically used for control of external hardware or for diagnostic purposes.

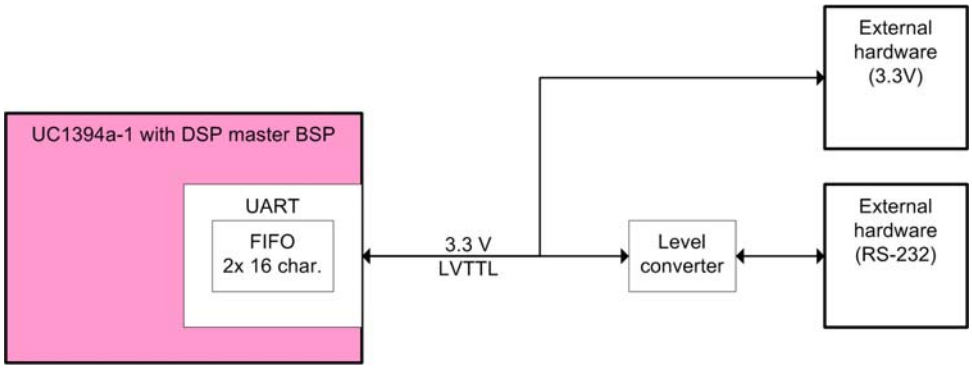


Figure 7: UART interface block diagram

5.4.1 UART Hardware Interface

The UART interface uses 2 data lines and 2 handshake lines.

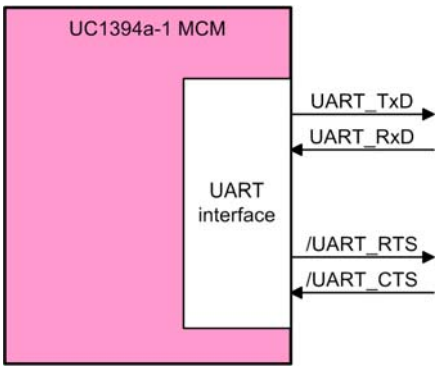


Figure 8: UART signals

Signal	MCM connector pin
UART_TxD	A17
UART_RxD	A18
/UART_RTS	A19
/UART_CTS	A20

Table 7: UART connector pin assignments

UART_TxD
Transmit data output of the UART interface. For RS-232 usage, this signal must be connected to a level converter. An example is shown in Figure 18.This signal is high when no data is transmitted.

UART_RxD
Receive data input of the UART interface. For RS-232 usage, this signal must be connected to a level converter. An example is shown in Figure 18.This input must be high when idle.

/UART_RTS
Handshake output of the UART interface. For RS-232 usage, this signal must be connected to a level converter. An example is shown in Figure 18. If RTS/CTS handshake is enabled (see chapter 5.4.2.2), RTS is controlled by the UART hardware. A logic low level on this signal indicates that the UART interface can accept more data. If RTS/CTS handshake is disabled, RTS is always active.

/UART_CTS
Handshake input of the UART interface. For RS-232 usage, this signal must be connected to a level converter. An example is shown in Figure 18. If RTS/CTS handshake is enabled (see chapter 5.4.2.2), the UART hardware only transmits characters if /UART_CTS is active (low). If RTS/CTS handshake is disabled, this input is ignored.

5.4.2 UART Programming

The UART interface is accessed through three FPGA registers:

Address	access direction	Register
020050_{16}	read	UART receive data
020050_{16}	write	UART transmit data
020051_{16}	read	UART control and status register

Using the UART interface is straightforward: Initialize the UART interface by writing the desired configuration (baud rate, handshake) to the UART control register. Then, data can be transferred using the transmitter and receiver ready flags and the UART data register.
Programming example:

```
#include "dsp_master_bsp.h" /* board support package definitions */

//initialize UART for 115200 baud and hardware handshake
```

```
UC1394A_UART_CTL = UC1394A_UART_CTL_BAUD_115200 | UC1394A_UART_CTL _HS_RTS_CTS;
...
//check if a character can be sent
if (UC1394A_UART_CTL & UC1394A_UART_CTL_TXRDY)
{
    //transmit a character
    UC1394A_UART_DATA= *pTxBuffer++;
...
```

5.4.2.1 UART Data Register (UART_DATA)

Description:
This register is used to transfer data from and to the UART interface. Data written to it will be transmitted on the UART interface. Received data can be read from this register. Before accessing this register, the status bits of the UART (TXRDY and RXRDY) should be checked.

Address:
020050_{16}

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
								UART_DATA							
r,00000000								r,w							

UART_DATA (Read Accesses Only)
8-bit receive data. Read accesses to this register read one character from the 16-bit receive FIFO. If the FIFO is empty, the last read value is repeated. Please check for available characters before reading this register.

```
#include "dsp_master_bsp.h" /* board support package definitions */

/* check for incoming characters */
if (UC1394A_UART_CTL & UC1394A_UART_CTL_RXDY)
{
    INT8U ucRxChar = UC1394A_UART_DATA;
    /* process incoming character */
}
```

UART_DATA (Write Accesses Only)
8-bit transmit data. The character that is written to this register is converted to serial format and transmitted on the TxD line, as soon as

- no other character is currently being transmitted and
- RTS/CTS Handshake is disabled or CTS is active (low).

Before writing to the transmit data register, software must check if the transmitter hardware can accept a new character:

```
#include "dsp_master_bsp.h" /* board support package definitions */

/* can I send now? */
if (UC1394A_UART_CTL & UC1394A_UART_CTL_TXDY)
{
    /* yes, send now */
    UC1394A_UART_DATA = '\n';
}
```

5.4.2.2 UART Control and Status Register (UART_CTL)

Description:
This register provides two function: It controls the baud rate at which the UART interface transmits and receives data and it provides status flags.

Address:
020051_{16}

Encoding:

15 14 13 12 11 10 9 8 7 6	5	4	3	2	1	0
RESERVED	OV	BD1	HS	BD0	RXRDY	TXRDY
r,0	r,wc,0	r,w,0	r,w,0	r,w,1	r,0	r,1

Transmit ready (TXRDY)
This bit indicates that the transmitter part of the UART is ready to accept a new data word to transmit. This bit must be read as '1' set before data is written to the UART_DATA register. See also the code example above (UART_DATA write access).

Receiver ready (RXRDY)
This bit indicates, that receive data is available in the UART_DATA register. This bit should be checked to poll for incoming data. See also the code example above (UART_DATA read access).

Baud rate (BD[1:0])
This bit controls the baud rate of the UART. Programming example:

```
#include "dsp_master_bsp.h" /* board support package definitions */

    /* initialize UART to use 115200 baud (and RTS/CTS handshake) */
    UC1394A_UART_CTL = UC1394A_UART_CTL_BAUD_115200 | UC1394A_UART_CTL_HS_RTS_CTS;
```

BD1	BD0	Baud rate
0	0	19200
0	1	115200 (default)
1	0	38400
1	1	reserved

Handshake (HS)
This bit controls, whether or not data transfers use the RTS/CTS handshake lines. If enabled, RTS/CTS handshake is controlled by the UART hardware. No further software intervention is required. Programming example:

```
#include "dsp_master_bsp.h" /* board support package definitions */

    /* initialize UART to use RTS/CTS handshake (and 115200 baud) */
    UC1394A_UART_CTL = UC1394A_UART_CTL_HS_RTS_CTS | UC1394A_UART_CTL_BAUD_115200;
```

HS	Handshake
0	none (default)
1	RTS/CTS handshake

Receiver overflow (OV)
This bit is set whenever a character from the RS-232 interface is received while the receive FIFO of the UART is full. Writing a '1' to this bit clears it. Programming example:

```
#include "dsp_master_bsp.h" /* board support package definitions */

    /* check for receiver overflow: */
    if (UC1394A_UART_CTL & UC1394A_UART_CTL_OV)
    {
        /* increment error counter */
        UartErrors++;
        /* clear the error flag */
        UC1394A_UART_CTL = UC1394A_UART_CTL | UC1394A_UART_CTL_OV;
...
```

OV	Meaning
0	normal operation
1	receiver overflow, at least one character got lost.

5.5 I/O Pins

Please note: With exception of the XF pin, all I/O pins of the MCM are implemented in the FPGA, whereas the (on-chip) GPIO pins of the DSP are not available on the UC1394a-1.
The available I/O pins are divided into three groups:

- bi-directional FPGA I/O: these I/O pins can be configured
- configuration inputs: these I/O pins can only be polled
- external flag (XF): this is a dedicated output of the DSP which can be controlled by BSET/BCLR instructions (see [1] or [2]).

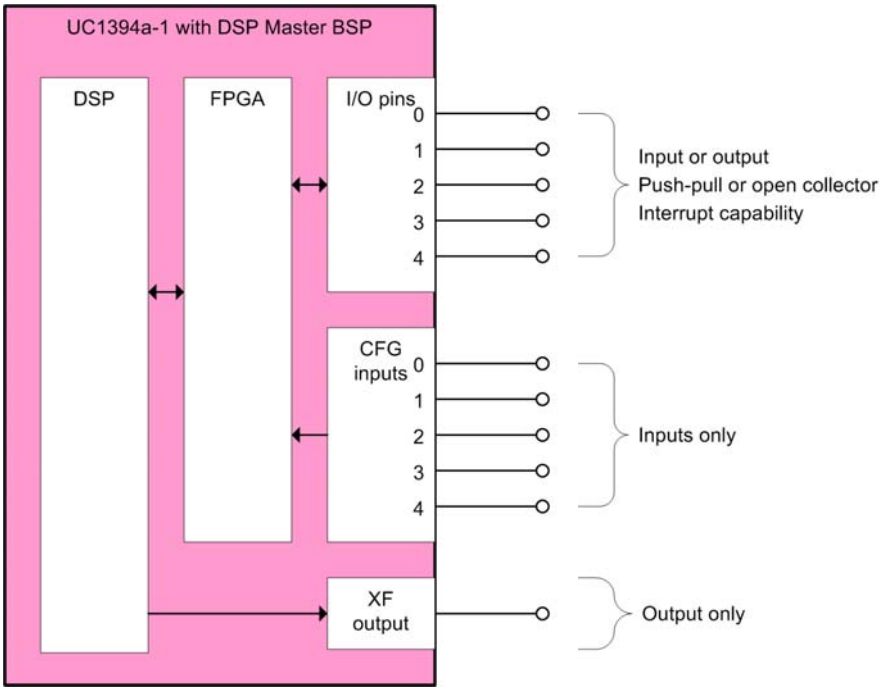


Figure 9: I/O pin block diagram

Each FPGA I/O pin can be individually configured to be an

- input
- push-pull output
- open drain output

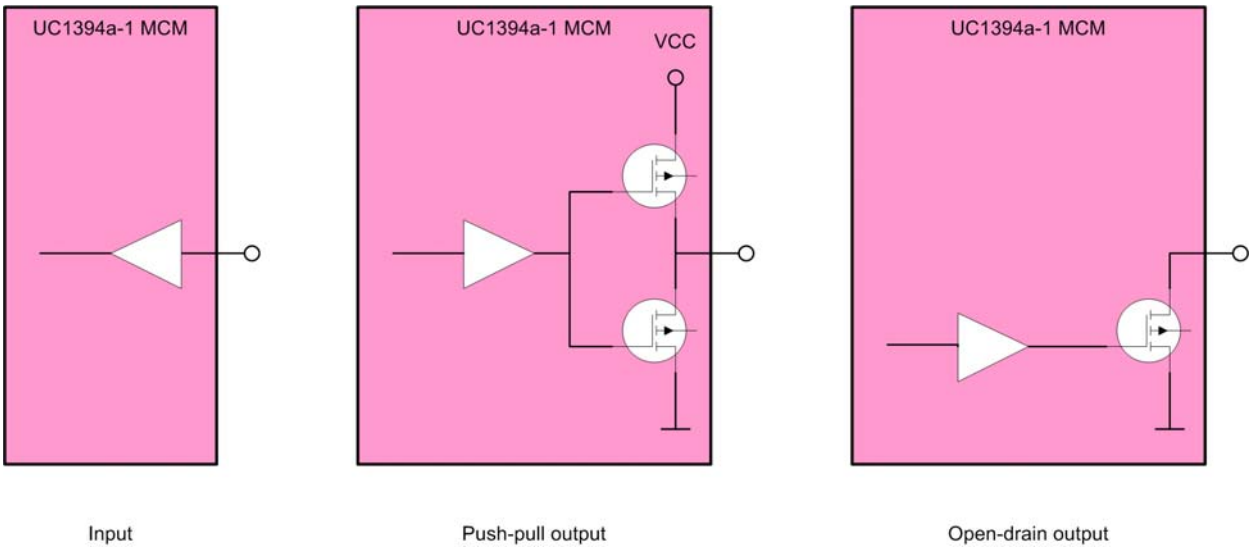


Figure 10: I/O pin configurations

5.5.1 I/O Pin Hardware Interface

Signal	MCM connector pin
I/O0	D19
I/O1	D20
I/O2	D21
I/O3	D22
I/O4	C27
CFG0	A12
CFG1	A13
CFG2	A14
CFG3	A15
CFG4	A16
XF	C26

Table 8: I/O pin assignments

5.5.2 I/O Pin Programming

Each FPGA I/O pin is controlled by an associated FPGA pin control register (PCR). If configured as an input, I/O pins can generate interrupts. These interrupts are OR-ed together and connected to the /INT1 input of the DSP.
All configuration inputs are read through the CFG register. These inputs can only be polled and have no interrupt capability.
The XF pin can be set and cleared in C using *asm* statements as shown below. Please note the whitespace before the XF instruction. It prevents that the BSET instruction is interpreted as a label.

```
//set XF pin to high (+3.3V)
asm(" BSET XF");
...
//set XF pin to low (0V)
asm(" BCLR XF");
```

5.5.2.1 Pin control registers (PCR)

Description:
These registers control operation of the I/O pins. Each register has an associated I/O pin, e.g. PCR 0 is associated with I/O0.

Address:
020080_{16} – 020084_{16}

Encoding:

15 14 13 12 11 10 9 8	7	6 5	4	3	2	1	0
RESERVED	IPOL	ISRC	IF	IE	O_MOD	DIR	DAT
r,00000000	r,w,0	r,w,00	r,wc,0	r,w,0	r,w,0	r,w,0	r,(w),0

Pin state (DAT)
For input pins, this bit reflects the current state of the pin (DAT read-only). For output pins, this bit controls the current state of the pin and can be read back.

Pin direction (DIR)
This bit controls the pin's direction.

DIR	direction
0	input
1	output

Output mode (O_MOD)
This bit field configures the behavior of the I/O pin when configured as an output. In input mode (DIR = 0), this bit field has no meaning.

O_MOD	output type
0	open collector (default)
1	push-pull (totem-pole) output

Interrupt enable (IE)
For input pins, this bit controls whether or not interrupts are generated by this pin. For output pins, this bit has no effect. If an interrupt is currently pending (IF = 1) and is disabled (IE = 0), it will become active as soon as IE is set to 1. Therefore, it might be necessary to clear previously pending interrupts before IE is set to 1.

IE	interrupts
0	disabled (default)
1	enabled

Interrupt flag (IF)
The interrupt flag signals, whether an interrupt is currently pending. The state of the IE bit has no influence on the IF bit. A pending interrupt can be cleared by writing a '1' to this bit.

Interrupt source (ISRC)
This bit field controls, how interrupts can be generated by this pin (if the pin is configured as an input (DIR = 0)). It is encoded as follows:

ISRC	interrupt source
00_2	No interrupts will be generated by this pin (default)
01_2	level triggered interrupt (see also IPOL bit)
10_2	edge triggered interrupt (see also IPOL bit)
11_2	delta interrupt (an interrupt will be generated by both, a falling or a rising edge on this pin)

Interrupt polarity bit (IPOL)
This pin determines the polarity of this pin, when used as a level or edge triggered interrupt.

IPOL	interrupts on
0	high level / rising edge (default)
1	low level / falling edge

together with the ISRC field, the following interrupt sources are available:

ISRC	IPOL	interrupt
00_2	0	No interrupts will be generated by this pin (default)
	1	
01_2	0	interrupts will be generated as long as this pin is low
	1	interrupts will be generated as long as this pin is high
10_2	0	interrupts will be generated on each falling edge on this pin
	1	interrupts will be generated on each rising edge on this pin
11_2	X	An interrupt will be generated by both, a falling or a rising edge on this pin

Programming examples:

```
#include "orstypes.h"       /* architecture independent data types */
#include "c5509.h"          /* DSP definitions */
#include "dsp_master_bsp.h" /* board support package definitions */
#include "misc.h"           /* module support lib functions */

void pin_examples(void)
{
    /* set up I/O Pin 0 as an input for polling */
    UC1394A_PCR0 = UC1394A_PCR_DIR_IN;

    /* set up I/O Pin 1 as an input with interrupts on either edge */
    UC1394A_PCR0 = (UC1394A_PCR_DIR_IN    |
                    UC1394A_PCR_ISRC_DELTA);
    /* install interrupt handler for I/O pin interrupts */
    C5xIntHook(C5509_INT1, IoPinISR);
    /* enable I/O pin interrupts on DSP level */
    C5xIntEnable(C5509_INT1);
    /* enable interrupts for I/O pin 1 */
    UC1394A_PCR1|= UC1394A_PCR_INT_ENABLE;
    /* disable interrupts for I/O pin 1 */
    UC1394A_PCR1&= ~UC1394A_PCR_INT_ENABLE;

    /* set up I/O pin 2 as a push-pull output */
    /* Note: When switching from input to output it is recommended to */
    /* set the pin state first and then to change direction */
    /* set state */
    if (OutputState == 1)
        UC1394A_PCR2|=  UC1394A_PCR_DAT;
    else
        UC1394A_PCR2&= ~UC1394A_PCR_DAT;
    /* change direction */
    UC1394A_PCR2|=  UC1394A_PCR_DIR_OUT;
    /* toggle output */
    UC1394A_PCR2^=  UC1394A_PCR_DAT;
    /* change back to input */
    UC1394A_PCR2&= ~UC1394A_PCR_DIR_OUT;
}

interrupt void IoPinISR(void)
{
    int iPinIdx;
    BOOL bInterruptPresent = TRUE;

    /* repeat as long as there is any interrupt. Otherwise, the */
    /* interrupt line stays asserted (low) and no further interrupts */
```

```
    /* can be triggered */
    while (bInterruptPresent == TRUE)
    {
        bInterruptPresent = FALSE;

        for (iPinIdx = 0; iPinIdx < 4; iPinIdx++)
        {
            /* check which pin generated the interrupt */
            if ((*(volatile INT16U *)&PCR0 + iPinIdx) & UC1394A_PCR_INT_FLAG)
            {
                /* clear pin's interrupt flag */
                (*(volatile INT16U *)&PCR0 + iPinIdx) = UC1394A_PCR_INT_FLAG;
                /* take the appropriate action */
                /* ... */
                /* cause pins to be checked for interrupts again */
                bInterruptPresent = FALSE;
            }
        }
    }
}
```

5.5.2.2 Configuration Register (CFG)

This register contains the state of the configuration inputs CFG[4:0]. The configuration inputs are used for configuring the operation mode in other UC1394a-1 based products. For the DSP Master BSP, these pins can simply be used as general-purpose inputs.

Address:
020042_{16}

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED											CFG				
r,000000000000											r				

Configuration (CFG):
This bit field contains the current state of CFG[4:0].
Programming example:

```
#include "dsp_master_bsp.h" /* board support package definitions */
int iPin;
/* poll all CFG pins */
for (iPin = 0; iPin < 4; iPin++)
if (UC1394A_CFG & (1 << iPin))
{
    /* pin is high */
    ...
}
else
{
    /* pin is low */
    ...
}
```

5.6 Other Interfaces Provided by FPGA Registers

5.6.1 System Control Register (SYS_CTL)

This register controls two functions:

- the red LED that is located on the UC1394a-1
- a software-triggered hardware reset

Several sources can drive the LED for diagnostic purposes. The LED can also be controlled by application software using the LED bit. When LEDSRC is set to its default value of zero, the LED can be controlled by the functions *LedOn()* and *LedOff()* of the module support library.

Address:
020040_{16}

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED								SWR	LED	RESERVED			LEDSRC		
r,0000000								w,0	r,w,0	r,000			r,w,000		

LED source selection (LEDSRC)
This bit field controls, which signal controls the LED: It is encoded as follows:

LEDSRC value	controlling source
000_2	manual control through LED bit (default)
001_2	LLC STAT0 pin
010_2	LLC register accesses
011_2	LLC datamover activity
100_2	interrupt line INT1
101_2	interrupt line INT2
110_2	interrupt line INT3
111_2	interrupt line INT4

LED control bit (LED)
This bit controls the LED when LEDSRC is set to manual LED control. Application software can use the LED as an optical indicator, e.g. for system status display. If this bit is set to '1', the red LED of the UC1394a-1 will light.

Software reset (SWR)
This bit can be used to reset the complete system by software. When this bit is set to 1, a hardware reset will be issued by pulling the /RESET_IN line low for a short time, so the system will behave exactly like it does when an external reset is asserted. Reading the SWR bit always returns '0'.

5.6.2 Watchdog Control Register (WDG)

The UC1394a-1 has a built-in watchdog timer, which can be used to recover from fatal conditions, such as deadlocks or software crashes. The watchdog timer, if enabled, must be reset periodically for normal operation. This is usually done by a function that is called periodically and which is vital for the user's application. In case of a fatal condition, the watchdog timer will no longer be reset, it will it time out and a system reset will be triggered. This is done by the FPGA pulling the /RESET_IN line low for at least 1µs, so the system will behave exactly like it does when an external reset is asserted.

Address:
020041_{16}

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED														WDE	WDR
r,00000000000000														r,(w),0	w,0

watchdog reset (WDR)
Each time this bit is written as '1', the watchdog timer is reset and the system operates normally. This bit has no effect, if the watchdog timer is disabled (WDE set to '0'). WDE is always read as '0'. The WDR bit must be written at least once per second to keep the watchdog permanently reset. The watchdog must be reset first, before it is enabled.

watchdog enable (WDE)
This bit field selects the source, from which the watchdog can be reset. It is encoded as:

WDE	watchdog status
0	watchdog disabled
1	manual reset by the WDR bit

Note: The WDE bit can only be set, but not reset. Once set, the WDE bit stays set for until the next hardware reset. This prevents accidental disabling of the watchdog. If you plan to use the watchdog, it must be periodically reset as long as the system is running (until power-off). The watchdog timer is disabled after reset, so if you don't plan to use it, no action is required. Before enabling the watchdog, it must be reset.
Programming example:

```
#include "dsp_master_bsp.h" /* board support package definitions */
#include "misc.h"           /* module support lib functions */

void main(void)
{
    Setup DSP();
    ...
    //reset and enable watchdog
    UC1394A_WDG_CTL = UC1394A_WDG_RESET;
    UC1394A_WDG_CTL = UC1394A_WDG_ENABLE;
    ...
//enter main loop
    while(1)
    {
        //reset watchdog
        UC1394A_WDG_CTL = UC1394A_WDG_RESET;
        ...
    }
}
```

5.6.3 Version Register (VER)

This register contains information about the FPGA version. It can be used by application software to check that the correct FPGA version is loaded and to check which FPGA revision is loaded.

Address:
020043_{16}

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VER								REV							
r								r							

FPGA revision (REV)
This bit field contains the current FPGA revision. The FPGA revision can be changed due to bug fixes or product enhancement.

FPGA version (VER):
This bit field identifies the current FPGA version. The version defines the functional behavior of the FPGA as well as the supported registers. The following FPGA versions are currently supported for the DSP master BSP:

FPGA version	applicable for
03_{16}	UC1394a-1 with a 144MHz TMS320VC5509
07_{16}	UC1394a-1 with a 200MHz TMS320VC5509A

Application software should check the version register after loading the FPGA.
Programming example:

```
#include "dsp_master_bsp.h" /* board support package + basic hardware def's */
#include "fpga_load.h"      /* FPGA loader */
```

```
// load FPGA from Flash
if (FpgaLoad ((INT32U*)UC1394A_FLASH_FPGA_CODE_BASE,
              UC1394A_FLASH_FPGA_CODE_LENGTH) != FPGA_SUCCESS)
{
    while (1); // stop (we do not have a LED as indicator yet)
}

//check for correct FPGA version
usFpgaVersion = (UC1394A_VERSION & UC1394A_VERSION_VER_MASK) >> 8;
if (bIs5509A)
{
    if (usFpgaVersion != 0x07)
        while(1); //stop (we probably do not have a LED as indicator yet)
}
else
{
    if (usFpgaVersion != 0x03)
        while(1); //stop (we probably do not have a LED as indicator yet)
}
```

6 Hardware Implementation Guidelines

This chapter shows the necessary connections for integrating the UC1394a-1 into a customized hardware environment. A complete wiring example is shown in chapter 6.6.

6.1 Power Supply

The UC1394-1 requires a stabilized 3.3 V supply. All of the supply and ground pins must be connected. The carrier PCB must have a ground plane with short connections to the MCM's ground pins. Decoupling capacitors of 100nF are recommended at each of the supply pins. Additional decoupling capacitors can be placed between the VCC and ground planes near the GND pins of the MCM. If the carrier PCB uses a voltage regulator to generate the +3.3 V, follow the instructions of the regulator's manufacturer for the type and value of the regulator's output capacitor(s).
If the carrier PCB is directly supplied with 3.3 V from a cable, a 10 .. 100µF capacitor in parallel with a 100nF capacitor is recommended at the point where the cable is connected.

6.2 IEEE1394 Interface

For connection to the IEEE1394 network, the 4-pin and 6-pin connectors defined in the standards [9], [10] should be used. The PCB traces to the connector (TPA and TPB signal pairs)

- must be kept as short as possible
- must be routed as a differential pair
- must have a differential impedance of 110±6Ω

Table 10 shows part number examples for both connector types.

Please note that the cable shield usage differs between the 4-pin and the 6-pin connector. See Figure 14 or Figure 15 and Figure 18 for connection details of the 6-pin and 4-pin connectors.

Signal	Pin	
	4-pin connector	6-pin connector
TPB-	1	3
TPB+	2	4
TPA-	3	5
TPA+	4	6
signal ground	connector shield	2
cable power	n/a	1
chassis ground[4]	n/a	connector shield

Table 9: Pinning of the IEEE1394 connectors

Connector type	Molex part No.
6-pin (IEEE1394-1995)	53462-xxx
6-pin with latch	55395-xxx
4-pin (IEEE1394a-2000)	54515-xxx

Table 10: IEEE1394 connector part numbers

[4] IEEE1394-1995 requires an isolation circuit such as shown in Figure 14 between cable shield and chassis ground, whereas IEEE1394a removed the requirement for an isolated cable shield connection.

When cable power is required, the 6-pin connector defined by IEEE1394-1995 must be used, as shown in Figure 11. Otherwise the 4-pin connector defined by IEEE1394a can be used (Figure 12). This connector is smaller and is often used in laptop computers. For industrial environment, 6-pin connectors with a robust case and latch are also available.
Figure 14 shows an example for supplying the MCM from IEEE1394, Figure 15 shows how to additionally supply power to IEEE1394. Further information about cable power usage can be found in [8], [9] and [10]. Connection details for the 4-pin connector are shown in Figure 18.

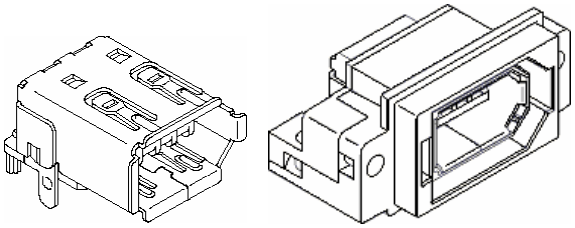


Figure 11: 6-pin IEEE1394 connectors

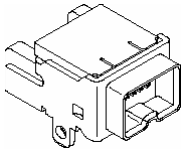
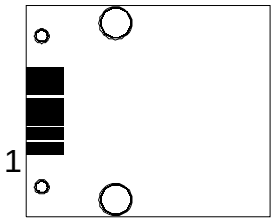


Figure 12: 4-pin IEEE1394a connector



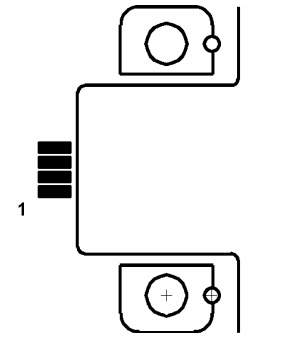


Figure 13: Pin numbering for 6-pin and 4-pin IEEE1394 connectors (top view)

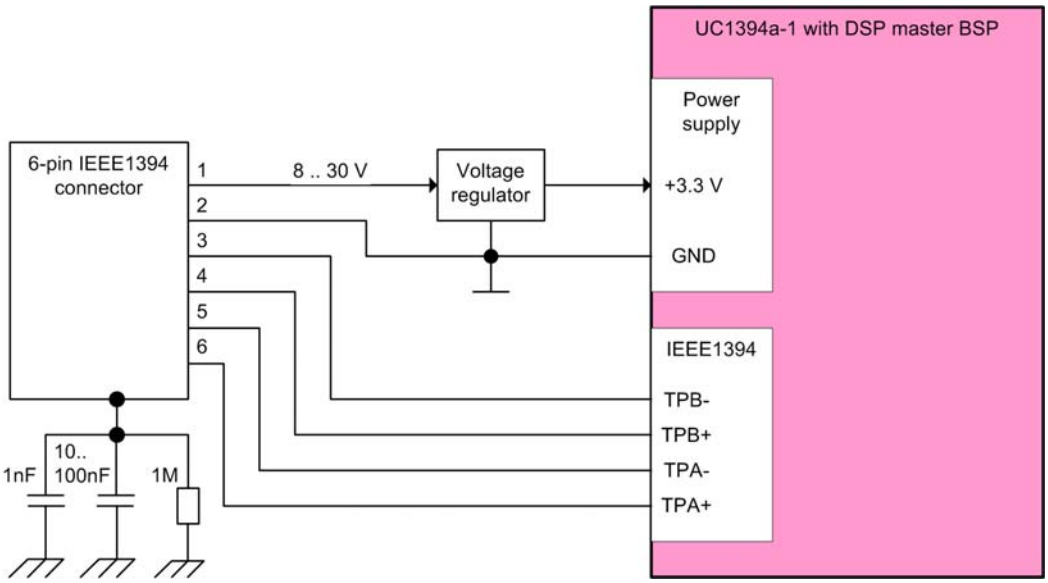


Figure 14: Supplying the MCM from IEEE1394

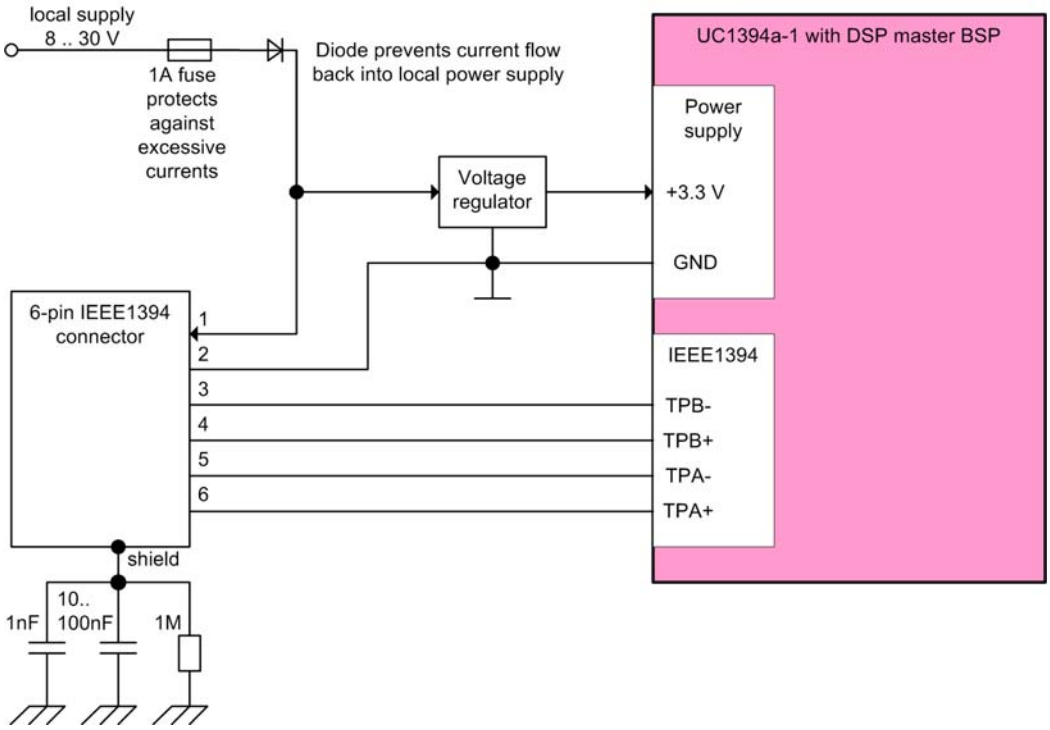


Figure 15: Supplying power to the IEEE1394 cable

6.3 RS-232 Level-Converter

Figure 16 shows a wiring example and Table 11 shows the required cable wiring . A detailed schematic example is shown in Figure 18. Using a 3.3V type is recommended.

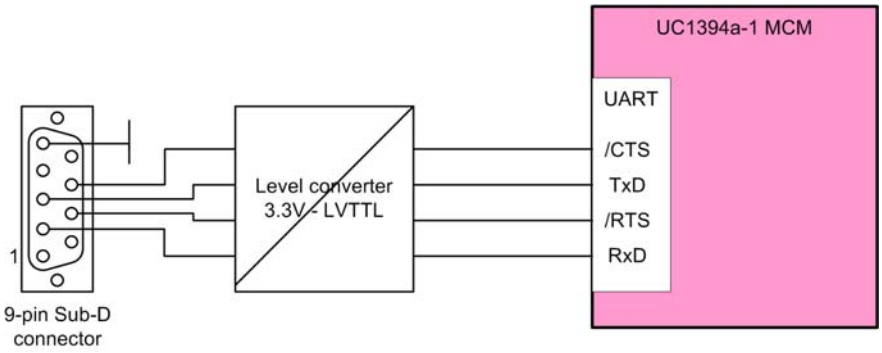


Figure 16: Wiring of the UART interface

Signal	Level converter		Host PC	
	Sub-D 9 pin	Sub-D 25 pin	Sub-D 9 pin	Sub-D 25 pin
RxD	2	3	3	2
TxD	3	2	2	3
RTS	7	4	8	5
CTS	8	5	7	4
GND	5	7	5	7

Table 11: Required cable connection to a host PC

6.4 JTAG Interface

The JTAG Interface is not needed during operation or configuration. However, it provides a possibility for debugging or for firmware updates in situations where the firmware loader doesn't

work. If the application permits, the JTAG signals of the DSP should be available in end-application environment for service purposes. The FPGA JTAG signals are not required. Figure 17 shows the wiring of the JTAG connector that can be used with the standard development tools.

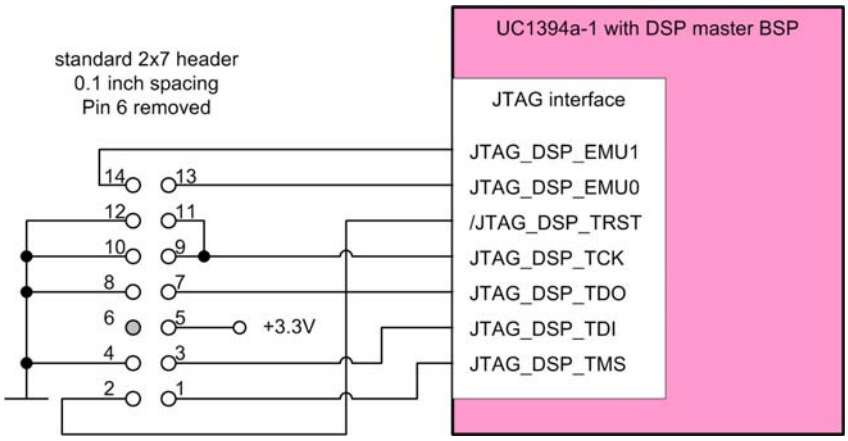


Figure 17: Wiring of the DSP JTAG interface

6.5 Unused Signals

Signals that are not used can be left unconnected. Most inputs have pull-up resistors or keeper circuits (see chapter 7.2 for details). For the McBSP signals, external pull-up resistors can be added to avoid unnecessary power consumption caused by floating inputs. The same applies to the USB interface, here a pull-up can be added to DP and a pull-down to DN.

6.6 Minimal Connection Example

Figure 18 shows the required connections for basic operation of the UC1394a-1 MCM, assuming that the following interfaces are being used:

- IEEE1394
- UART interface
- peripheral interface

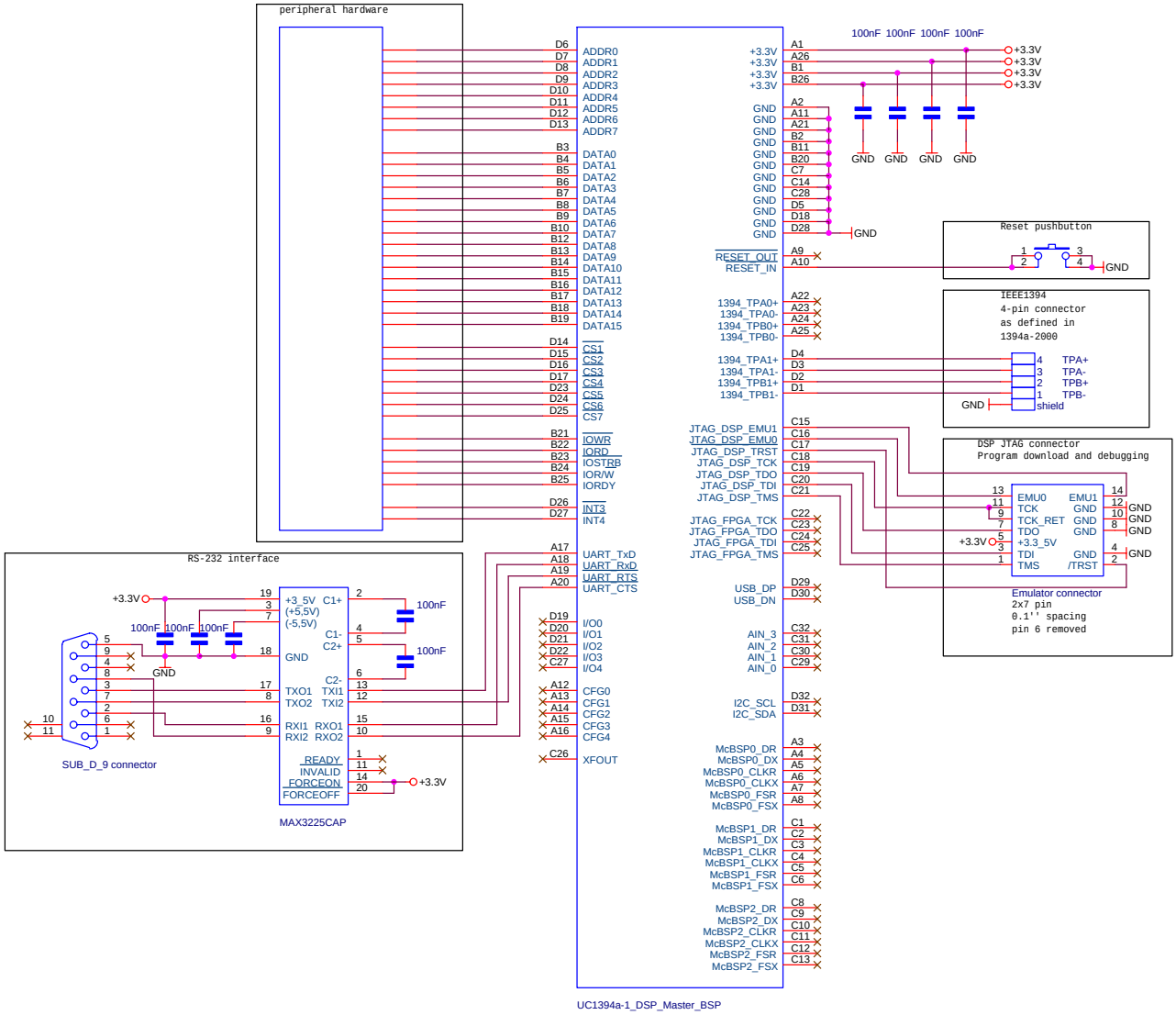


Figure 18: Required connections

7 Technical Data

7.1 Signal Overview and Connector Pinout Tables

This chapter defines all signals that are supported by the UC1394a-1 with DSP master BSP. The shaded entries in Table 12 show signals that are always available, independent of the board support package, whereas all other signals are specific to the DSP master BSP.

Pin	Connector A	Connector B	Connector C	Connector D	Pin
1	+3.3V	+3.3V	McBSP1_DR	TPB1-	1
2	GND	GND	McBSP1_DX	TPB1+	2
3	McBSP0_DR	DATA0	McBSP1_CLKR	TPA1-	3
4	McBSP0_DX	DATA1	McBSP1_CLKX	TPA1+	4
5	McBSP0_CLKR	DATA2	McBSP1_FSR	GND	5
6	McBSP0_CLKX	DATA3	McBSP1_FSX	ADDR0	6
7	McBSP0_FSR	DATA4	GND	ADDR1	7
8	McBSP0_FSX	DATA5	McBSP2_DR	ADDR2	8
9	/RESET_OUT	DATA6	McBSP2_DX	ADDR3	9
10	/RESET_IN	DATA7	McBSP2_CLKR	ADDR4	10
11	GND	GND	McBSP2_CLKX	ADDR5	11
12	CFG0	DATA8	McBSP2_FSR	ADDR6	12
13	CFG1	DATA9	McBSP2_FSX	ADDR7	13
14	CFG2	DATA10	GND	/CS1	14
15	CFG3	DATA11	JTAG_DSP_EMU1	/CS2	15
16	CFG4	DATA12	JTAG_DSP_EMU0	/CS3	16
17	UART_TxD	DATA13	/JTAG_DSP_TRST	/CS4	17
18	UART_RxD	DATA14	JTAG_DSP_TCK	GND	18
19	/UART_RTS	DATA15	JTAG_DSP_TDO	I/O0	19
20	/UART_CTS	GND	JTAG_DSP_TDI	I/O1	20
21	GND	/IOWR	JTAG_DSP_TMS	I/O2	21
22	TPA0+	/IORD	JTAG_FPGA_TCK	I/O3	22
23	TPA0-	/IOSTRB	JTAG_FPGA_TDO	/CS5	23
24	TPB0+	IOR/W	JTAG_FPGA_TDI	/CS6	24
25	TPB0-	IORDY	JTAG_FPGA_TMS	/CS7	25
26	+3.3V	+3.3V	XFOUT	/INT3	26
27			I/O4	/INT4	27
28			GND	GND	28
29			AIN0	USB_DP	29
30			AIN1	USB_DN	30
31			AIN2	I2C_SDA	31
32			AIN3	I2C_SCL	32

= Pin function independent of the DSP Master BSP

Table 12: Pinout sorted by pins

Signal	Pin number
+3.3V	A1
	A26
	B1
	B26
GND	A2
	A11
	A21
	B2
	B11
	B20
	C7
	C14
	C28
	D5
	D18
	D28
/RESET_IN	A10
/RESET_OUT	A9

Table 13: Power supply and reset signals

Signal	Pin number
ADDR0	D6
ADDR1	D7
ADDR2	D8
ADDR3	D9
ADDR4	D10
ADDR5	D11
ADDR6	D12
ADDR7	D13
DATA0	B3
DATA1	B4
DATA2	B5
DATA3	B6
DATA4	B7
DATA5	B8
DATA6	B9
DATA7	B10
DATA8	B12
DATA9	B13
DATA10	B14
DATA11	B15
DATA12	B16
DATA13	B17
DATA14	B18
DATA15	B19
/CS1	D14
/CS2	D15
/CS3	D16

Signal	Pin number
/CS4	D17
/CS5	D23
/CS6	D24
/CS7	D25
/IOWR	B21
/IORD	B22
/IOSTRB	B23
IOR/W	B24
IORDY	B25
/INT3	D26
/INT4	D27

Table 14: Peripheral interface signals

Signal	Pin number
I/O0	D19
I/O1	D20
I/O2	D21
I/O3	D22
I/O4	C27
CFG0	A12
CFG1	A13
CFG2	A14
CFG3	A15
CFG4	A16
XFOUT	C26

Table 15: I/O pin signals

Signal	Pin number
UART_TxD	A17
UART_RxD	A18
/UART_RTS	A19
/UART_CTS	A20

Table 16: UART interface signals

Signal	Pin number
JTAG_DSP_EMU1	C15
JTAG_DSP_EMU1	C16
/JTAG_DSP_TRST	C17
JTAG_DSP_TCK	C18
JTAG_DSP_TDO	C19
JTAG_DSP_TDI	C20
JTAG_DSP_TMS	C21

Table 17: DSP JTAG signals

Signal	Pin number
TPA0+	A22
TPA0-	A23
TPB0+	A24
TPB0-	A25
TPA1+	D4
TPA1-	D3
TPB1+	D2
TPB1-	D1

Table 18: IEEE1394 signals

Signal	Pin number
McBSP0_DR	A3
McBSP0_DX	A4
McBSP0_CLKR	A5
McBSP0_CLKX	A6
McBSP0_FSR	A7
McBSP0_FSX	A8
McBSP1_DR	C1
McBSP1_DX	C2
McBSP1_CLKR	C3
McBSP1_CLKX	C4
McBSP1_FSR	C5
McBSP1_FSX	C6
McBSP2_DR	C8
McBSP2_DX	C9

Signal	Pin number
McBSP2_CLKR	C10
McBSP2_CLKX	C11
McBSP2_FSR	C12
McBSP2_FSX	C13

Table 19: McBSP signals

Signal	Pin number
AIN0	C29
AIN1	C30
AIN2	C31
AIN3	C32

Table 20: ADC signals

Signal	Pin number
I2C_SDA	D31
I2C_SCL	D32

Table 21: I^2C signals

Signal	Pin number
USB_DP	D29
USB_DN	D30

Table 22: USB signals

7.2 Individual Signal Description

+3.3V

Power supply for the UC1394a-1. These pins provide the power supply for the UC1394a-1. All necessary internal voltages are generated from this voltage. Please refer to chapter 0 for voltage limits and recommended operating conditions. The power supply lines must be properly stabilized by decoupling capacitors as described in chapter 6.1.

direction	polarity	built-in termination	handling when not used
n/a (power)			

GND

These pins are the power supply and signal ground pins of the UC1394a-1. They should be directly connected to the ground plane of the carrier PCB.

direction	polarity	built-in termination	handling when not used
n/a (power)			

/RESET_IN

Reset input and software reset output. If this pin is set to logic low level, the UC1394a-1 is reset. This pin should only be driven by an open-drain output or a pushbutton connected to ground. When software or the watchdog timer of the UC1394a-1 trigger a reset, /RESET_IN is pulled low for at least 1μs. The minimum input pulse width for externally applied resets is also 1 μs. A power-on reset circuit is provided on the UC1394a-1 so that /RESET_IN can be left open if not used.

direction	polarity	built-in termination	handling when not used
bi-directional	active-low	4.7 kΩ pull-up	leave open

/RESET_OUT

This is the active low reset output line of the UC1394a-1. It allows external hardware devices to be reset and exactly be started together with the UC1394a-1. This output is pulled low whenever a system reset is active, thus

- after power-on
- when /RESET_IN is externally pulled low
- when the firmware of the UC1394a-1 performs a software reset (FPGA register)
- when the watchdog timer of the UC1394a-1 triggers a reset

This output will stay low for 140ms .. 300ms after the reset condition is removed. After that time, the output is pulled high.

direction	polarity	built-in termination	handling when not used
output	active-low	none	leave open

CFG0 .. CFG4

These pins can be used as general purpose inputs. The current state of CFG[4:0] can be read from a FPGA register (see chapter 5.5.2.2).

direction	polarity	built-in termination	handling when not used
input	active-high	pull-up (FPGA)	leave open

TPA0+, TPA0-, TPB0+, TPB0-

IEEE1394 signals of port 0. They must be connected as defined in [9] or [10]. See also chapter 6 for connection examples.

direction	polarity	built-in termination	handling when not used
bi-directional	n/a	110Ω differential	leave open

TPA1+, TPA1-, TPB1+, TPB1-
IEEE1394 signals of port 1. They must be connected as defined in [9] or [10]. See also chapter 6 for connection examples.

direction	polarity	built-in termination	handling when not used
bi-directional	n/a	110Ω differential	leave open

DATA0 .. DATA15
These are the bi-directional data lines of the peripheral interface. DATA0 .. DATA15 are only driven during write cycles to the peripheral interface. When DATA[15:0] are not driven, bus-hold circuits keep these signal at the previous logic level and keep them from floating.

direction	polarity	built-in termination	handling when not used
bi-directional	active-high	bus-holder (FPGA)	leave open

ADDR0…ADDR7
These are the address bus output lines of the UC1394a-1 peripheral interface. They are driven during read and write cycles of the peripheral interface. While the peripheral interface is idle, bus-hold circuits keep these signal at the previous logic level and keep them from floating.

direction	polarity	built-in termination	handling when not used
tri-state output.	active-high	bus-holder (FPGA)	leave open

/CS1…/CS7
These are the seven active low chip select output lines of the UC1394a-1 peripheral interface. They pre-select, which external peripheral component has to be accessed during a UC1394a-1 peripheral access. /CS[7:1] are always driven. During a peripheral access, one of /CS[7:1] is active (low) while the other chip select lines stay high.

direction	polarity	built-in termination	handling when not used
output	active-low	n/a	leave open

/IORD and /IOWR
These are the active low read strobe (/IORD) and I/O write strobe (/IOWR) output lines of the UC1394a-1 peripheral interface. They indicate a dedicated I/O read cycle (/IORD) or a dedicated I/O write cycle (/IOWR) of the UC1394a-1 peripheral interface. In case of a read cycle, data is latched at the rising edge of /IORD. Write data is valid prior the rising edge of /IOWR. /IORD and /IOWR are generated from the signals IOR/W and /IOSTRB and have the same timing as the /IOSTRB signal. They are useful to connect Intel compatible peripheral devices to the board, using the control signals /CSn (chip select), /IORD (I/O read) and /IOWR (I/O write).

direction	polarity	built-in termination	handling when not used
output	active-low	n/a	leave open

IOR/W and /IOSTRB
These are the read/write (IOR/W) and the active low I/O strobe (/IOSTRB) output lines of the peripheral interface. They indicate a dedicated I/O read cycle (IOR/W=high and /IOSTRB=low) or a dedicated I/O write cycle (IOR/W=low and /IOSTRB=low) of the UC1394a-1 peripheral interface. These signals allow to connect Texas Instruments or Motorola compatible peripheral devices to the board, using the signals /CSn (chip select), IOR/W (I/O read/write) and /IOSTRB (I/O strobe). The processor latches read data at the rising edge of /IOSTRB. Peripherals usually latch write data at the rising edge of /IOSTRB.

direction	polarity	built-in termination	handling when not used
output	active-low	n/a	leave open

/INT3…/INT4:
These are the two maskable interrupt input lines of the UC1394a-1 peripheral interface. They can be used as interrupts or for DMA synchronization. Interrupts are triggered on the falling edge of the interrupt signal. The /INT[4:3] signals are routed to the processor's interrupt lines /INT[4:3] respectively. Pull-up resistors for these signals are implemented in the FPGA.

direction	polarity	built-in termination	handling when not used
input	active-low	pull-up (FPGA)	leave open

IORDY:
This pin is the active high ready input of the UC1394a-1 peripheral interface. peripheral accesses the IORDY input is sampled by the processor. If IORDY is sampled low, the current access is extended by one EMIF clock and then IORDY is sampled again. If IORDY is sampled high, the access cycle completes. If the connected peripherals do not use the IORDY signal, IORDY should be left unconnected. The necessary pull up resistor is provided on the UC1394a-1. Please note: The IORDY input should only be used when the wait state generator (see chapter 5.1.1) does not provide enough cycle time.

direction	polarity	built-in termination	handling when not used
input	active-high	pull-up (FPGA)	leave open

I/O0 .. /O4
General purpose I/O pins. They can be used as

- inputs
- open-drain outputs
- push-pull outputs

Please refer to chapter 5.5 for details. IO[4:0] have pull-up resistors implemented in the FPGA.

direction	polarity	built-in termination	handling when not used
bi-directional	active-high	pull-up (FPGA)	leave open

XFOUT
This pin is directly connected to the DSP of the UC1394a-1. It can be used as a general purpose output pin. The default state of XFOUT after reset is high.

direction	polarity	built-in termination	handling when not used
output	active-high	n/a	leave open

UART_TxD
Transmit data output of the UART interface. For RS-232 usage, this signal must be converted to the appropriate level as shown in chapter 6.3.

direction	polarity	built-in termination	handling when not used
output	active-high	n/a	leave open

UART_RxD
Receive data input of the UART interface. For RS-232 usage, this signal must be driven by a level converter as shown in chapter 6.3. /UART_RxD has a pull-up implemented in the FPGA.

direction	polarity	built-in termination	handling when not used
input	active-high	pull-up (FPGA)	leave open

/UART_RTS
Handshake output of the UART interface. For RS-232 usage, this signal must be converted to the appropriate level as shown in chapter 6.3.

direction	polarity	built-in termination	handling when not used
output	active-low	n/a	leave open

/UART_CTS
Handshake input of the UART interface. For RS-232 usage, this signal must be driven by a level converter as shown in chapter 6.3. /UART_RTS has a pull-up implemented in the FPGA.

direction	polarity	built-in termination	handling when not used
input	active-low	pull-up (FPGA)	leave open

McBSP0_DR, McBSP1_DR, McBSP2_DR
Data receive input of the respective McBSP interface. This pin can also be used as a general purpose input. The McBSP signals are directly connected to the DSP. Please refer to [5] for a detailed description.

direction	polarity	built-in termination	handling when not used
input	programmable	none	leave open or pull-up

McBSP0_DX, McBSP1_DX, McBSP2_DX,
Data transmit output of the respective McBSP interface. This pin can also be used as a general purpose output. The McBSP signals are directly connected to the DSP. Please refer to [5] for a detailed description.

direction	polarity	built-in termination	handling when not used
output	programmable	n/a	leave open

McBSP0_CLKR, McBSP1_CLKR, McBSP2_CLKR,
Data receive clock of the respective McBSP interface. This pin can also be used as a general purpose I/O pin. The McBSP signals are directly connected to the DSP. Please refer to [5] for a detailed description.

direction	polarity	built-in termination	handling when not used
bi-directional	programmable	none	leave open or pull-up

McBSP0_CLKX, McBSP1_CLKX, McBSP2_CLKX,
Data transmit clock of the respective McBSP interface. This pin can also be used as a general purpose I/O pin. The McBSP signals are directly connected to the DSP. Please refer to [5] for a detailed description.

direction	polarity	built-in termination	handling when not used
bi-directional	programmable	none	leave open or pull-up

McBSP0_FSR, McBSP1_FSR, McBSP2_FSR,
Receive frame synchronization I/O of the respective McBSP interface. This pin can also be used as a general purpose I/O pin. The McBSP signals are directly connected to the DSP. Please refer to [5] for a detailed description.

direction	polarity	built-in termination	handling when not used
bi-directional	programmable	none	leave open or pull-up

McBSP0_FSX, McBSP1_FSX, McBSP2_FSX,
Transmit frame synchronization I/O of the respective McBSP interface. This pin can also be used as a general purpose I/O pin. The McBSP signals are directly connected to the DSP. Please refer to [5] for a detailed description.

direction	polarity	built-in termination	handling when not used
bi-directional	programmable	none	leave open or pull-up

USB_DP, USB_DN
USB data lines. They must be connected according to the USB standard. Please refer to [5] for a detailed description.

direction	polarity	built-in termination	handling when not used
bi-directional	n/a	1K5 pull-up on DP	leave open

I2C_SDA

I^2C data line. This line is a bi-directional signal with open-drain output and a 4.7kΩ pull-up resistor. Please refer to [5] for a detailed description.

direction	polarity	built-in termination	handling when not used
bi-directional	active-high	4.7 kΩ pull-up	leave open

I2C_SCL

I^2C clock line. This line is an open-drain output with a 4.7kΩ pull-up resistor. Please refer to [5] for a detailed description.

direction	polarity	built-in termination	handling when not used
output	active-high	4.7 kΩ pull-up	leave open

AIN0 .. AIN3

ADC inputs of the UC1394a-1 MCM DSP. These pins are directly connected to the DSP

direction	polarity	built-in termination	handling when not used
input	n/a	none	leave open

JTAG_DSP_EMU1, JTAG_DSP_EMU0, /JTAG_DSP_TRST, JTAG_DSP_TCK, JTAG_DSP_TDO, JTAG_DSP_TDI, JTAG_DSP_TMS,

These pins are used for debugging and software download. and can be used with a JTAG emulator. The JTAG_DSP_EMU0 and JTAG_DSP_EMU1 inputs have a 4.7kΩ pull-up resistor.

direction	polarity	built-in termination	handling when not used
depending on specific signal			leave open

JTAG_FPGA_TCK, JTAG_FPGA_TDO, JTAG_FPGA_TDI, JTAG_FPGA_TMS

These pins can be used for downloading FPGA code. They are used with Xilinx download cables together with the FPGA development option.

direction	polarity	built-in termination	handling when not used
depending on specific signal			leave open

7.3 Dimensions of the UC1394a-1

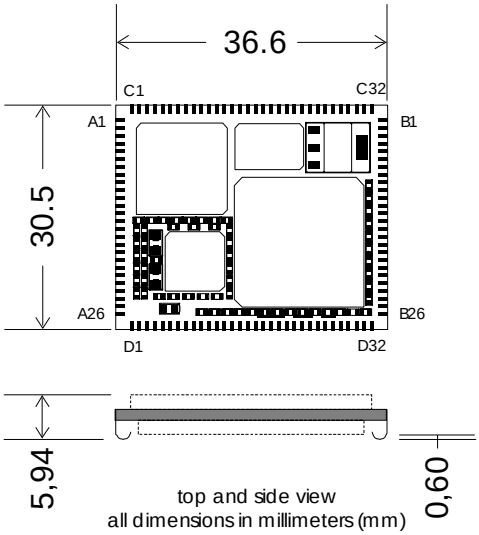


Figure 19: Dimensions of the UC1394a-1 (including connector pins)

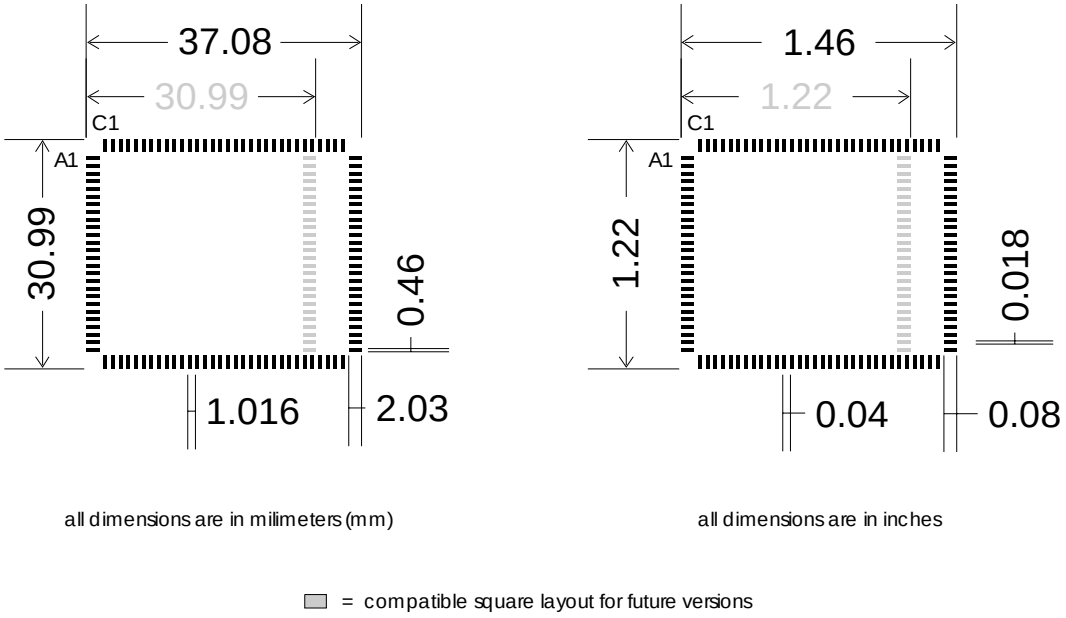


Figure 20: Recommended PCB footprint of the UC1394a-1

Please note: The PCB area below the UC1394a-1 should not be used for components.

7.4 Environmental Conditions

7.4.1 Storage

The UC1394a-1 can be stored in its original packaging for one year at the conditions given in chapters 7.4.2 and 7.4.3.

7.4.2 Ambient Humidity

Parameter	Max
storage, non condensing	90%
operating, non condensing	85%

7.4.3 Ambient Temperature

Parameter	Min	Max
storage temperature	-25°C	+85°C
operating temperature	0°C	60°C

Please note:
The ambient temperature can be higher than 60°C if the FPGA case temperature is limited to 80°C by appropriate cooling methods, such as ventilation, heat sinks and good thermal design of the carrier PCB.

7.5 Soldering Process

The UC1394a-1 is designed to be placed and soldered like an integrated circuit, allowing mass production. The UC1394a-1 can be soldered using vapor phase or reflow processes, just like BGA packages.
Figure 21 shows an example of a temperature curve that was measured during production of the UC1394a-1 using a lead-free reflow process. Please note that the UC1394a-1 starting with S/N 012231 is RoHS compliant and therefore produced using a lead-free process. For mounting older versions of the UC1394a-1, please contact Orsys.
Unless otherwise noted on the packaging, baking is required before soldering.

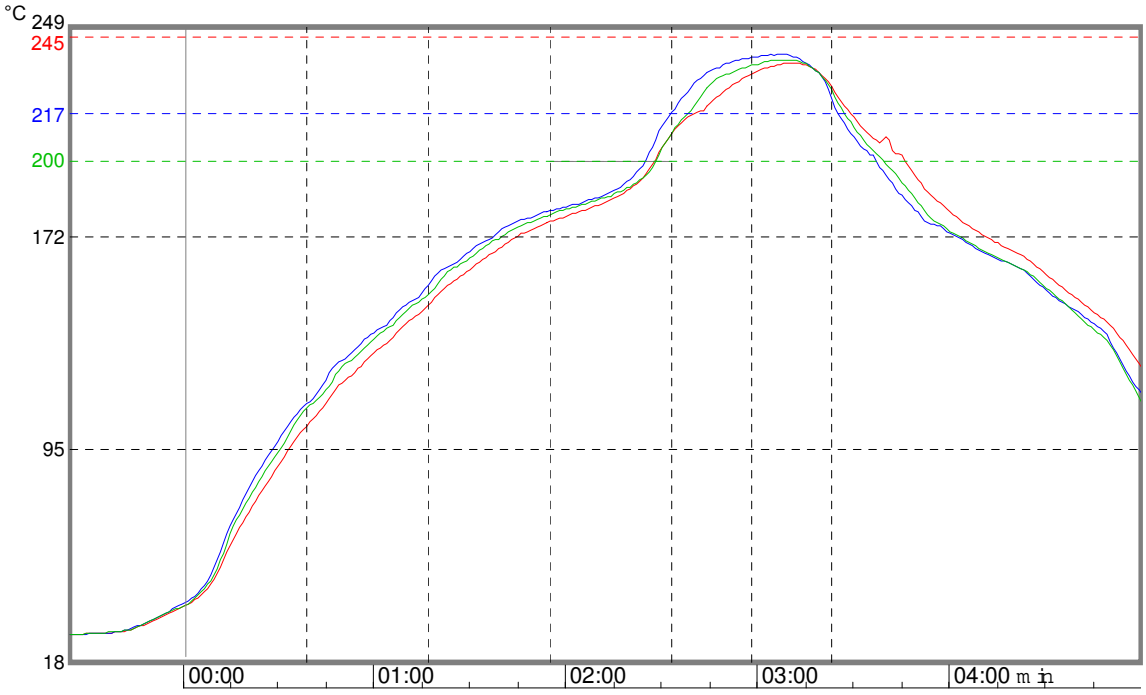


Figure 21: Soldering temperature example

7.6 Power Requirements

the power requirements shown below are valid for both, 144MHz and 200MHz versions of the UC1394a-1. The higher clock frequency of the 200MHz TMS320VC5509A is compensated by the newer, power saving architecture, so that the overall power consumption is the same or slightly less than the one of a 144MHz UC1394a-1 with a TMS320VC5509 DSP.

Parameter	Min	Typ	Max
supply voltage	3.2 V	3.3 V	3.6 V
current consumption		450 mA	

Table 23: Power requirements

7.7 Signal Levels and Loads

In general, all digital logic pins can be used with 3.3V signal levels. Detailed specifications for each signal group are listed throughout this chapter.

CAUTION:

Applying more than 3.6V to inputs that are not 5V tolerant will damage the device!

7.7.1 FPGA Signals

A group of signals are connected to the FPGA. They use a 5V input tolerant LVTTL I/O standard. Please refer to [11] for a detailed description of the FPGA's LVTTL signal levels. The following signals include:

Interface	Signal	Direction
Configuration inputs	CFG[4:0]	input
I/O pins	I/O[4:0]	input/output
Peripheral interface	DATA[15:0]	input/output
	ADDR[7:0]	output
	/CS[7:1]	output
	/IORD	output
	/IOWR	output
	IOR/W	output
	/IOSTRB	output
	RDY	input
	/INT[4:3]	input
UART	UART_TXD	output
	UART_RXD	input
	UART_RTS	output
	UART_CTS	input

Parameter	Value
Compatible I/O standards	5V TTL 3.3V LVTTL 2.5V CMOS
High input level	2.0V .. 5.5V
Low input level	-0.5V .. 0.8V
High output level	min. 2.4V
Low output level	max. 0.4V
maximum DC load	2 mA

Table 24: Signal levels and loads for FPGA signals

7.7.2 Reset Signals

/RESET_IN is connected to the on-board power on reset circuit. It can furthermore be pulled low by the FPGA. Therefore, /RESET_IN is a bi-directional signal. A 4.7kΩ pull-up resistor gives a default level of 3.3V (at no load).

Parameter	Value
Compatible I/O standards	3.3V LVTTL 2.5V CMOS
High input level	2.3V .. 3.3V
Low input level	0V .. 1.0V

Table 25: /RESET_IN signal levels

/RESET_OUT is a push-pull output which is directly driven from the on-board reset generator.

Parameter		Value
Compatible I/O standards		3.3V LVTTL 2.5V CMOS
High output level		min. 2.64 V
Low output level		max. 0.3 V
maximum DC load	low	1.2 mA
maximum DC load	high	-0.5 mA

Table 26: /RESET_OUT signal levels

7.7.3 DSP Signals

A group of (digital) lines are directly connected to the DSP. Please refer to [5] for a detailed description.
These signals are:

Interface	Signal	Direction
McBSP	all Signals	(input /output)
IO	XFOUT	(output)
I²C	I2C_SDA	(input /output)
	I2C_SCL	(output)

Parameter	Value
Compatible I/O standards	3.3V LVTTL 2.5V CMOS
High input level	2.0V .. 3.6V
Low input level	-0.5V .. 0.8V
High output level	min. 2.4V
Low output level	max. 0.4V
maximum DC load	see [1], [2]

Table 27: Signal level and loads for the DSP signals

7.7.4 Analog Inputs

These signals are directly connected to the DSP. Please refer to [5] for a detailed description.

Interface	Signal	Direction
ADC	AIN[3:0]	(input)

Parameter	Value
minimum input voltage	0V
maximum input voltage	3.3V

Table 28: Allowed input voltage range for the ADC inputs

7.7.5 Other Signals

The remaining signals are intended to be used with the appropriate interfaces only. For example, the FPGA JTAG signals should only be used with programming equipment from XILINX, so that correct signal levels and loads are guaranteed.

The interfaces for use with dedicated equipment are:

- IEEE1394 signals
- USB
- JTAG signals for the DSP
- JTAG signals for the FPGA

7.8 Peripheral Interface Timing

The peripheral interface timing is based on the TMS320VC5509 EMIF timing. The timing diagrams and tables below show the timing parameters for the default EMIF settings, as set up by the module support library.

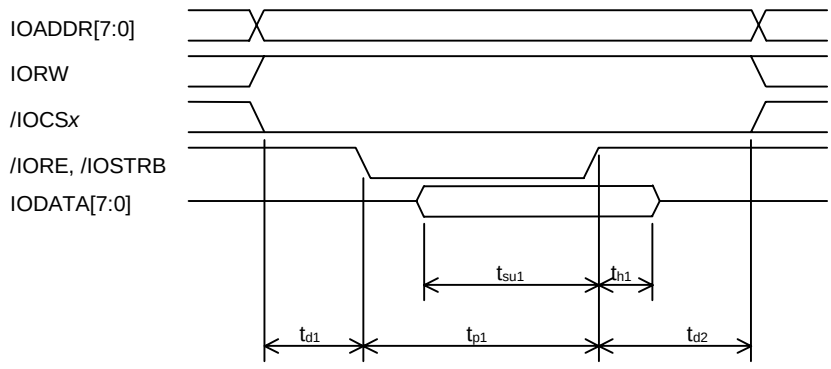


Figure 22: Peripheral interface read timing

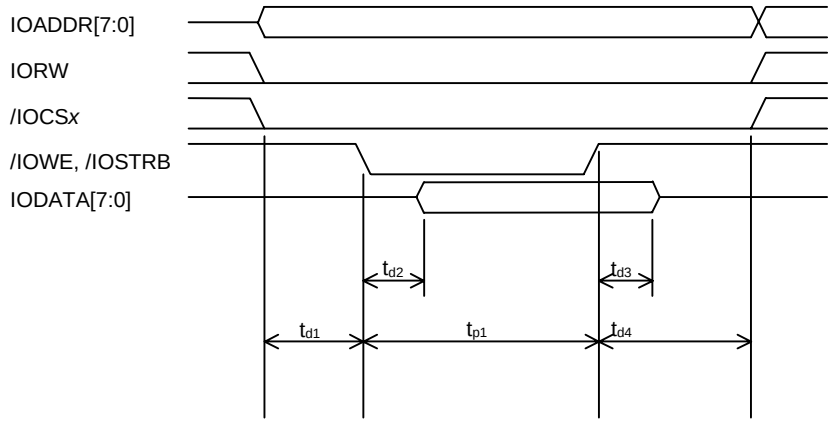


Figure 23: Peripheral interface write timing

7.8.1 Timings for a 200MHz TMS320VC5509A

Measurement conditions

- FPGA V7.02
- UC1394a-1 set up by GEL file or the DSP boot loader and module support library function *SetupDsp*
- Wait state generator at default setting (0 wait states)

parameter		value	
name	description	min	max
t_{d1}	strobe signals low after control signals valid	4 ns	
t_{p1}	strobe signals active	55 ns	65 ns
t_{su1}	data valid before strobe high	20 ns	
t_{h1}	data valid after strobe high	0 ns	
t_{d2}	control signals valid after strobe signals high	5 ns	
t_{su2}	ARDY valid before strobe high	50 ns	

Figure 24: Peripheral interface read timing (200MHz)

parameter		value	
name	description	min	max
t_{d1}	strobe signals low after control signals valid	4 ns	
t_{p1}	strobe signals active	55 ns	65 ns
t_{d2}	data valid after strobe low		10 ns
t_{d3}	data valid after strobe high	5 ns	10 ns
t_{d4}	control signals valid after strobe signals high	5 ns	
t_{su2}	ARDY valid before strobe high	50 ns	

Figure 25: Peripheral interface write timing (200MHz)

To relax these timings for slower peripheral components, the peripheral accesses can be extended up to 205 ns strobe time by using the wait state generator (see chapter 5.1.1). Each wait state adds 2 CPU clocks (5 ns at 200 MHz) to timing parameter t_{p1}.

7.8.2 Timings for a 144MHz TMS320VC5509

Measurement conditions

- FPGA V3.09
- UC1394a-1 set up by GEL file or the DSP boot loader and module support library function *SetupDsp*

parameter		value	
name	description	min	max
t_{d1}	strobe signals low after control signals valid	5 ns	
t_{p1}	strobe signals active	41 ns	
t_{su1}	data valid before strobe high	30 ns	
t_{h1}	data valid after strobe high	0 ns	
t_{d2}	control signals valid after strobe signals high	5 ns	
t_{su2}	ARDY valid before strobe high	50 ns	

Figure 26: Peripheral interface read timing (144MHz)

parameter		value	
name	description	min	max
t_{d1}	strobe signals low after control signals valid	5 ns	
t_{p1}	strobe signals active	41 ns	
t_{d2}	data valid after strobe low		10 ns
t_{d3}	data valid after strobe high	5 ns	
t_{d4}	control signals valid after strobe signals high	5 ns	
t_{su2}	ARDY valid before strobe high	50 ns	

Figure 27: Peripheral interface write timing (144MHz)

To relax these timings for slower peripheral components, the EMIF settings can be modified by increasing the default settings for setup, strobe and hold cycles for read or write accesses. Each additional clock adds one CPU clock period (6.94 ns for a CPU clock of 144MHz) to the timings as listed below.

- Increasing the setup time increases t_{d1} for read and write accesses, allowing more time for chip select, address and direction decoding.
- Increasing the number of strobe cycles increases tp1 for read and write accesses. For read accesses this allows longer access times ($t_{acc} = t_{p1} – t_{su1}$) of the peripheral component. For write accesses, this allows longer setup times for the peripheral component.
- Increasing the hold time increases t_{d2} for read accesses and t_{d4} for write accesses. This gives the peripheral component more time to finish the access

7.9 Reset Timing

parameter	value	
	min	max
/RESETIN input pulse width	1µs	
/RESETOUT pulse width	140ms	280ms

Table 29: Reset timing

7.10 I/O Pin Timings

For the I/O pins no timings are defined, because the I/O pins mainly depend on software processing.

8 Glossary

ADC	analog to digital converter
BSP	board support package, a specific combination of software and FPGA design that adds certain functions to the UC1394a-1.
Byte	A data word consisting of 8 bits of data. This terminology appears in the IEEE1394 standards and is also commonly used
CCS	Code Composer Studio: An integrated development environment for digital signal processors provided by Texas Instruments.
Configuration ROM	a well-defined location that must be implemented in each device. It contains information about the device, such as capabilities and supported software protocols
Doublet	A data word consisting of 16 bits of data. This terminology appears in the IEEE1394 standards
Endiannes	The order, in which data words are assembled to larger entities (i.e. bytes to quadlets). The IEEE1394 standards use big endian notation, so the most significant part comes first and is located at lower addresses.
firmware	The combination of software and FPGA code that is installed on the UC1394a-1. This software will be booted from flash memory at power up or system reset.
FPGA	field programmable gate array
I^2C	=IIC = inter-IC-communication. A two wire interface between integrated circuits, such as EEPROM's, temperature sensors, etc.
IEEE1394	Standard for a high speed serial bus. Also known as Fire Wire or i-Link, which are the trademarks of Apple inc. and SONY respectively.
LED	light emitting diode
LLC	link layer controller (for IEEE1394)
LSB	least significant bit or byte
LSW	least significant word
MCM	multi-chip module
MSB	most significant byte
MSW	most significant word
McBSP	multi-channel buffered serial port: A peripheral interface of the TMS320VC5509 DSP of the UC1394a-1
n.a.	not available
open collector output	an output that drives only the logic 0 state to GND
open drain output	an output that drives only the logic 0 state to GND
Phy	physical layer transceiver (for IEEE1394)
push-pull output	an output that drives both states, logic 0 to VCC and logic 1 to GND
Quadlet	A data word consisting of 32 bits of data. This terminology appears in the IEEE1394 standards
root directory	an entry in the configuration ROM that contains general information about the device.
RSV	reserved
RTC	real time clock
TBC	to be changed. This is subject to change, so do not rely on it
TBD	to be defined. The value for this is not yet defined.
USB	universal serial bus; an interface for peripheral devices

9 Literature References

Further information that is not covered in this user's guide can be found in the documents listed below. References to this list are given in square brackets throughout this document. The documents are listed by title, author and literature number or file name

[1] *TMS320VC5509 Fixed-Point Digital Signal Processor Data Manual*, TI, SPRS163
[2] *TMS320VC5509A Fixed-Point Digital Signal Processor Data Manual*, TI, SPRS205
[3] *TMS320VC5509 Digital Signal Processor Silicon Errata*, TI, SPRZ006
[4] *TMS320C55x DSP CPU Reference Guide*, TI, SPRU371
[5] *TMS320C55x DSP Peripherals Reference Guide*, TI, SPRU317
[6] *TMS320C55x Assembly Language Tools User's Guide*, TI, SPRU280
[7] *TMS320C55x Optimizing C/C++ Compiler User's Guide*, TI, SPRU281
[8] *FireWire System architecture* by Don Anderson, Mind Share Inc., ISBN 0-201-48535-x
[9] *IEEE Standard for a High Performance Serial Bus* ,IEEE, Std 1394-1995
[10] *IEEE Standard for a High Performance Serial Bus—Amendment 1*, IEEE, Std 1394a-2000
[11] *IEEE Standard for a Control and Status Registers (CSR) Architecture for Microcomputer Buses*, IEEE, Std 1212-2001
[12] *Spartan-II 2.5V FPGA Family: DC and Switching Characteristics*, Xilinx, DS001-3, www.xilinx.com
[13] *TSB12LV32 IEEE1394 and P1394a Compliant General-Purpose Link-Layer Controller*, TI, SPRU317
[14] *User Guide IEEE1394 embedded API*, Orsys, emb_1394_API_UG.pdf
[15] *DSP Development Kit User's Guide*, Orsys, DSP_DevKit_UG.pdf
[16] *FlashBurn homepage*, Software Design Solutions Inc., www.softwaredesignsolutions.com